

Algorithms, data structures and computability

Michel Wermelinger

30 August 2025

CONTENTS

1	Introduction	3
1.1	What to expect	3
1.1.1	Content	3
1.1.2	Challenges	5
1.1.3	Previous knowledge	6
1.2	How to study	7
1.2.1	Book	7
1.2.2	Mindset	9
1.2.3	Time allocation	10
1.2.4	Study techniques	11
1.3	Software	12
1.4	Summary	13
1.4.1	Notebooks	14
1.4.2	Common commands	14
1.4.3	Command mode	15
1.4.4	Edit mode	16
2	Numbers and sequence	17
2.1	Numbers	18
2.1.1	Integers and real numbers	18
2.1.2	int and float literals	19
2.1.3	Mistakes	20
2.2	Arithmetic operations	21
2.2.1	On real numbers	21
2.2.2	On integers	22
2.2.3	On int and float	23
2.2.4	On float	23
2.2.5	On int	24
2.2.6	Mistakes	25
2.3	Expressions	26
2.3.1	Mistakes	28
2.4	Assignments	28
2.4.1	Algorithms	29
2.4.2	Names	30

2.4.3	Mistakes	31
2.5	Functions in mathematics	33
2.5.1	Example	35
2.5.2	Algorithms	36
2.5.3	Mistakes	37
2.6	Functions in Python	38
2.6.1	Documentation	40
2.6.2	Mistakes	41
2.7	Complexity	44
2.7.1	Constant complexity	44
2.7.2	Linear complexity	45
2.7.3	Mistakes	47
2.8	Run-times	48
2.8.1	Checking growth rates	50
2.9	Summary	53
2.9.1	Python	53
2.9.2	Data types	53
2.9.3	Functions	55
2.9.4	Complexity	56
3	Booleans and selection	57
3.1	Booleans	57
3.1.1	The Boolean ADT	58
3.1.2	Using Booleans	60
3.1.3	The <code>bool</code> type	60
3.1.4	Mistakes	61
3.2	Decision problems	62
3.2.1	Problem definition	62
3.2.2	Problem instances	63
3.2.3	Algorithm	65
3.2.4	Complexity	65
3.2.5	Code and tests	66
3.2.6	Performance	66
3.3	Boolean expressions	67
3.3.1	Comparisons	68
3.3.2	Mistakes	70
3.4	Classification problems	71
3.4.1	Problem definition and instances	72
3.4.2	Algorithm	73
3.4.3	Complexity	77
3.4.4	Code	77
3.4.5	Tests	79
3.4.6	Performance	80
3.4.7	Mistakes	80
3.5	Practice	81
3.5.1	Phone calls	81
3.5.2	Maximum	82
3.5.3	Leap year	83

3.6	Summary	84
3.6.1	Booleans	84
3.6.2	Solving problems	85
3.6.3	Classification problems	85
4	Sequences and iteration	87
4.1	The Sequence ADT	88
4.1.1	Inspecting sequences	88
4.1.2	Creating sequences	90
4.2	Strings	92
4.2.1	Literals	92
4.2.2	Inspecting strings	94
4.2.3	Creating strings	98
4.3	Iteration	101
4.3.1	For-loops	101
4.3.2	While-loops	104
4.3.3	Repeat-loops	105
4.3.4	Nested loops	106
4.4	Linear search	108
4.4.1	Finding characters	108
4.4.2	Valid password	111
4.5	Tuples and tables	114
4.5.1	Literals and operations	114
4.5.2	Mistakes	115
4.5.3	Tables	117
4.5.4	Iterating	119
4.6	Lists	121
4.6.1	Modifying sequences	121
4.6.2	Creating lists	123
4.6.3	Mistakes	125
4.6.4	Modifying lists	126
4.6.5	Mistakes	128
4.7	Reversal	129
4.7.1	Problem definition	129
4.7.2	Problem instances	130
4.7.3	Algorithm	131
4.7.4	Complexity	133
4.7.5	Code	133
4.7.6	Tests	134
4.7.7	Performance	134
4.8	Optional practice	138
4.8.1	DNA	138
4.8.2	Minimum	138
4.8.3	Lexicographic comparison	139
4.8.4	Palindrome	139
4.8.5	Mode	139
4.8.6	Images	139
4.9	Summary	143

4.9.1	Sequence operations	144
4.9.2	IPython	145
4.9.3	Python	145
4.9.4	Problems	147
4.9.5	Testing	147
4.9.6	Complexity	148
5	TMA 01 part 1	149
5.1	Problem definition	149
5.1.1	Problem and output names	150
5.1.2	Inputs and outputs	150
5.1.3	Preconditions	151
5.1.4	Postconditions	151
5.1.5	Test table	152
5.2	Algorithms and complexity	152
5.2.1	Linear search	153
5.2.2	Complexity	154
5.3	Coding style	154
5.3.1	The Zen of Python	155
5.3.2	Linters	156
6	Implementing sequences	159
6.1	Defining data types	160
6.1.1	Data structure	161
6.1.2	Functions	161
6.1.3	Classes	162
6.1.4	Mistakes	165
6.2	Static arrays	170
6.2.1	Variables and assignments	172
6.2.2	The <code>StaticArray</code> class	174
6.2.3	Testing methods	176
6.3	Developing data types	178
6.3.1	Abstract classes	179
6.3.2	Testing data types	181
6.4	Bounded sequences	183
6.4.1	Outlining algorithms	184
6.4.2	Insertion	185
6.4.3	The <code>BoundedSequence</code> class	186
6.5	Dynamic arrays	189
6.5.1	The <code>DynamicArray</code> class	190
6.5.2	Tests	191
6.6	Using dynamic arrays	192
6.6.1	The <code>ArraySequence</code> class	192
6.7	Linked lists	195
6.7.1	Traversing a linked list	195
6.7.2	Inserting an item	196
6.7.3	The <code>LinkedSequence</code> class	200
6.7.4	Linked list v. array	203

6.8	Summary	204
6.8.1	Classes	204
6.8.2	Data structures	205
6.8.3	Complexity	205
6.8.4	Python	206
7	Stacks and queues	207
7.1	Stacks	207
7.1.1	The stack ADT	208
7.1.2	Implementing with an array	209
7.1.3	Implementing with a linked list	211
7.2	Using stacks	213
7.2.1	Balanced brackets	213
7.2.2	Postfix expressions	217
7.3	Queues	218
7.3.1	Queues	218
7.3.2	Queues with Python lists	219
7.3.3	Queues with linked lists	220
7.3.4	Using queues	222
7.4	Priority queues	223
7.4.1	With dynamic arrays: version 1	224
7.4.2	With dynamic arrays: version 2	226
7.4.3	With a linked list	228
7.4.4	Min-priority queues	228
7.5	Summary	229
7.5.1	Stacks	229
7.5.2	Queues	230
7.5.3	Priority queues	230
8	Unordered collections	233
8.1	Maps	233
8.1.1	The map ADT	234
8.1.2	Using maps	234
8.1.3	Lookup tables	235
8.2	Dictionaries	238
8.2.1	Mistakes	240
8.2.2	Using dictionaries	242
8.3	Hash tables	244
8.3.1	With separate chaining	244
8.3.2	Hash functions	247
8.3.3	Unhashable values	249
8.3.4	Complexity	250
8.3.5	Implementation	250
8.4	Sets	253
8.4.1	The set ADT	253
8.4.2	Sets in Python	254
8.4.3	Implementing sets	256
8.4.4	Using sets	257

8.5	Summary	258
8.5.1	Maps and dictionaries	258
8.5.2	Lookup and hash tables	259
8.5.3	Sets	260
9	Practice 1	261
9.1	Pangram	262
9.1.1	Problem definition	263
9.1.2	Algorithm and complexity	263
9.1.3	Code and tests	263
9.2	Election	264
9.2.1	Problem definition	264
9.2.2	Algorithm and complexity	265
9.2.3	Code and tests	265
9.3	Voucher	265
9.3.1	Algorithm and complexity	266
9.3.2	Code and tests	267
9.4	Trains	267
9.4.1	Problem definition	268
9.4.2	Algorithm and complexity	269
9.4.3	Code and tests	269
9.5	SMS	269
9.5.1	First approach	270
9.5.2	Second approach	272
10	TMA 01 part 2	273
10.1	Using collections	273
10.1.1	Ordered collections	274
10.1.2	Unordered collections	274
10.2	Algorithms	275
10.3	Coding style	275
10.4	Python	276
10.4.1	Language constructs	276
10.4.2	Built-in types	276
10.4.3	Standard library	278
10.4.4	Other	278
10.4.5	M269 Library	278
11	Exhaustive search	279
11.1	Linear search, again	280
11.1.1	Basic search	280
11.1.2	Simultaneous and successive searches	282
11.1.3	Sorted candidates	283
11.2	Factorisation	284
11.2.1	Make candidates explicit	285
11.2.2	Compute solutions	287
11.2.3	Sort candidates	288
11.2.4	Prime numbers	290
11.3	Constraint satisfaction	291

11.3.1	Problem	292
11.3.2	Algorithm and complexity	292
11.3.3	Code and performance	293
11.3.4	Don't generate equivalent candidates	294
11.3.5	Reduce the range	295
11.3.6	Compute part of a candidate	296
11.3.7	Improved code and performance	296
11.4	Searching permutations	297
11.4.1	Problem	298
11.4.2	Algorithm	300
11.4.3	Complexity	300
11.4.4	Code	302
11.5	Searching subsets	303
11.5.1	Problem	304
11.5.2	Algorithm	305
11.5.3	Complexity	306
11.5.4	Code	307
11.6	Practice	308
11.6.1	Duplicates	308
11.6.2	Subset sum	309
11.6.3	Substring	310
11.6.4	PIN	310
11.7	Summary	310
11.7.1	Problems	311
11.7.2	Complexities	311
11.7.3	Reducing the search space	312
11.7.4	Python	312
12	Recursion	313
12.1	The factorial function	314
12.1.1	Definition and algorithm	314
12.1.2	Code	315
12.1.3	The call stack	316
12.2	Recursion on integers	318
12.2.1	Algorithm	318
12.2.2	Recursive definition	319
12.3	Length of a sequence	320
12.3.1	Recursive definition	320
12.3.2	Code	321
12.3.3	Mistakes	322
12.4	Inspecting sequences	323
12.4.1	Maximum	323
12.4.2	Membership	324
12.4.3	Indexing	326
12.5	Creating sequences	327
12.5.1	Prepend	327
12.5.2	Linear search	327
12.5.3	Append and insert	329

12.6	Avoiding slicing	329
12.6.1	Problem definition	330
12.6.2	Recursive definition	330
12.6.3	Code	331
12.7	Multiple recursion	333
12.7.1	Dividing the input	334
12.7.2	Designing multiple recursion	335
12.8	Summary	336
13	Divide and conquer	339
13.1	Decrease by one	340
13.1.1	Factorial	340
13.1.2	Sequence length	341
13.2	Decrease by half	342
13.2.1	Problem	342
13.2.2	Algorithm	343
13.2.3	Complexity	344
13.2.4	Code and performance	345
13.3	Variable decrease	346
13.3.1	Problem	346
13.3.2	Algorithm	347
13.3.3	Complexity	347
13.4	Binary search	348
13.4.1	Recursive, with slicing	349
13.4.2	Recursive, without slicing	350
13.4.3	Iterative	352
13.5	Binary search variants	353
13.5.1	Transition	354
13.5.2	Right number in the right place	357
13.6	Divide and conquer	357
13.6.1	Complexity	357
13.7	Summary	360
13.7.1	Complexity	360
13.7.2	Binary search	361
14	Sorting	363
14.1	Preliminaries	364
14.1.1	Problem	364
14.1.2	Problem instances	365
14.1.3	Algorithms	367
14.1.4	Sorting in Python	367
14.2	Bogosort	368
14.3	Insertion sort	369
14.3.1	Recursive version	369
14.3.2	Iterative version	370
14.3.3	Complexity	372
14.3.4	Performance	372
14.4	Selection sort	373

14.4.1	Recursive version	373
14.4.2	Iterative version	374
14.4.3	Code	375
14.4.4	Select largest	377
14.5	Merge sort	377
14.5.1	Algorithm	377
14.5.2	Complexity	380
14.5.3	Code and performance	380
14.6	Quicksort	382
14.6.1	Algorithm	382
14.6.2	Complexity	384
14.6.3	Code and performance	385
14.6.4	In-place version	387
14.7	Quicksort variants	387
14.7.1	Three-way quicksort	387
14.7.2	Quickselect	388
14.8	Pigeonhole sort	389
14.8.1	Comparison sort complexity	390
14.8.2	Algorithm	390
14.8.3	Complexity	391
14.8.4	Code and tests	391
14.8.5	Performance	393
14.9	Summary	393
14.9.1	Algorithms	394
14.9.2	Complexities	394
15 TMA 02 part 1		397
15.1	Brute-force search	397
15.1.1	Problem type	397
15.1.2	Candidates	398
15.1.3	Generate	398
15.1.4	Test	399
15.1.5	Solutions	399
15.1.6	Algorithm	399
15.1.7	Complexity	400
15.1.8	Performance	400
15.2	Divide and conquer	400
15.2.1	Complexity	401
16 Rooted trees		403
16.1	Binary tree	404
16.1.1	Terminology	405
16.1.2	ADT and data structure	406
16.2	Algorithms on trees	409
16.2.1	Divide and conquer	409
16.2.2	Arm's-length recursion	411
16.3	Traversals	412
16.3.1	Depth-first search	413

16.3.2	Pre-order traversal	414
16.3.3	In-order traversal	415
16.3.4	Post-order traversal	416
16.3.5	Breadth-first search	417
16.4	Binary search trees	419
16.4.1	Search	420
16.4.2	Add node	422
16.4.3	Remove node	423
16.5	Balanced trees	426
16.5.1	Complexity of search	427
16.5.2	Balanced trees	428
16.5.3	Checking balance	429
16.6	Heapsort	431
16.6.1	Binary heap	432
16.6.2	Inserting items	433
16.6.3	Removing the root	434
16.6.4	Complexity	435
16.6.5	Heaps in Python	436
16.7	Summary	437
16.7.1	Rooted trees	437
16.7.2	Binary trees	438
16.7.3	Binary search trees	438
16.7.4	Heaps	439

17 Graphs 1 441

17.1	Modelling with graphs	442
17.1.1	Exercises	446
17.2	Basic concepts	447
17.2.1	On nodes and edges	447
17.2.2	On graphs	448
17.2.3	Special graphs	450
17.2.4	ADT	452
17.3	Edge list representation	453
17.3.1	Exercises	454
17.4	Adjacency matrix representation	454
17.4.1	Exercises	456
17.5	Adjacency list representation	457
17.5.1	Exercises	458
17.6	Classes for graphs	459
17.6.1	The <code>DiGraph</code> class	460
17.6.2	The <code>UndirectedGraph</code> class	463
17.6.3	Special graphs	465
17.6.4	Random graphs	471
17.7	Traversing a graph	475
17.7.1	First algorithm	475
17.7.2	Complexity	476
17.7.3	Code and tests	477
17.7.4	Second algorithm	478

17.8	Breadth- and depth-first search	485
17.8.1	Breadth-first search	486
17.8.2	Depth-first search	487
17.8.3	Tests	488
17.8.4	Comparison	496
17.9	Summary	498
17.9.1	Terminology	499
17.9.2	Data structures	499
17.9.3	Traversals	500
17.9.4	Python	501
18	Greed	503
18.1	Interval scheduling	505
18.1.1	The greedy approach	506
18.1.2	Greedy choices	507
18.1.3	Algorithm	511
18.2	Weighted graphs	513
18.2.1	Data structures	515
18.2.2	Classes	516
18.3	Minimum spanning tree	521
18.3.1	First algorithm	522
18.3.2	Second algorithm	525
18.3.3	Code	526
18.4	Shortest paths	529
18.4.1	Algorithm	529
18.4.2	Code	531
18.4.3	Applications	538
18.5	Summary	539
18.5.1	Weighted graphs	539
18.5.2	Python	540
19	Practice 2	541
19.1	Jousting	542
19.1.1	Exercises	542
19.2	Dot product	543
19.2.1	Exercises	544
19.3	Beams	545
19.3.1	Exercises	545
19.4	Up and down	546
19.4.1	Exercises	547
19.5	A knight goes places	548
19.5.1	Exercises	549
20	TMA 02 part 2	551
20.1	Using graphs	551
20.1.1	Modelling with graphs	551
20.1.2	Algorithms	552
20.2	Applying greed	552
20.2.1	Approaches	552

20.2.2	Correctness	553
20.3	Python	554
20.3.1	Language constructs	554
20.3.2	Standard library	554
20.3.3	M269 library	554
21 Graphs 2		557
21.1	Undirected graph components	558
21.1.1	Problem definition and instances	559
21.1.2	Algorithm and complexity	560
21.1.3	Code and tests	560
21.2	Directed graph components	562
21.2.1	Problem and instances	563
21.2.2	Algorithm and complexity	564
21.2.3	Code and tests	566
21.3	Topological sort	567
21.3.1	Problem	567
21.3.2	Algorithm and code	569
21.3.3	Complexity	571
21.3.4	Exercises	571
21.4	State graphs	573
21.4.1	Problem	573
21.4.2	Graph	575
21.4.3	Code	576
21.4.4	Complexity	580
21.5	Practice	580
21.5.1	Rook's moves	581
21.5.2	Islands	582
21.6	Summary	583
22 Backtracking		585
22.1	Generate sequences	586
22.1.1	Recursive generation	587
22.1.2	Accept partial candidates	590
22.2	Prune the search space	591
22.2.1	Local and global constraints	592
22.2.2	Avoid visits	594
22.3	Trackword	595
22.3.1	The problem	596
22.3.2	Candidates and extensions	596
22.3.3	The constraints	598
22.3.4	Template	600
22.4	Optimise	600
22.4.1	The problem	601
22.4.2	Keep the best	601
22.4.3	Avoid worse candidates	603
22.4.4	Start well	604
22.5	Back to the TSP	605

22.5.1	The main function	607
22.5.2	The value function	608
22.5.3	Checking the constraints	609
22.5.4	The backtracking function	610
22.6	Generate subsets	611
22.7	Back to the knapsack	613
22.7.1	The problem	614
22.7.2	The value function	614
22.7.3	The constraints functions	615
22.7.4	The backtracking function	616
22.7.5	The main function	617
22.7.6	Sort extensions	619
22.8	Summary	621
23	Dynamic Programming	623
23.1	Fibonacci	624
23.1.1	Recursive	624
23.1.2	Top-down	626
23.1.3	Bottom-up	629
23.1.4	With arrays	630
23.1.5	A graph perspective	632
23.2	Longest common subsequence	633
23.2.1	Recursive	634
23.2.2	Top-down	636
23.2.3	Recursive with indices	637
23.2.4	Top-down with matrix	638
23.2.5	Bottom-up	640
23.2.6	Complexity and run-time	641
23.3	Knapsack	642
23.3.1	Recursive	643
23.3.2	Common subproblems	645
23.3.3	Top-down and bottom-up	646
23.3.4	Complexity	648
23.4	Summary	648
24	Practice 3	651
24.1	Safe places	652
24.1.1	Exercises	652
24.2	Extra staff	653
24.2.1	Exercises	654
24.3	Borrow a book	655
24.3.1	Exercises	656
24.4	Levenshtein distance	656
24.4.1	Exercises	657
24.5	Higher and higher	660
24.5.1	Exercises	660
25	TMA 03 part 1	663
25.1	Problems on graphs	663

25.1.1	Starting with the graph	663
25.1.2	Starting with the general problem	664
25.2	Backtracking	665
25.2.1	Constraints on sequences	665
25.2.2	Best sequence	666
25.2.3	Constraints on sets	668
25.2.4	Best set	669
25.3	Dynamic programming	671
26	Complexity classes	675
26.1	Tractable and intractable problems	676
26.1.1	Problem complexity	676
26.1.2	Tractable problems	676
26.1.3	Intractable problems	677
26.1.4	The twilight zone	679
26.2	The P and NP classes	680
26.2.1	SAT	680
26.2.2	Class P	681
26.2.3	Class NP	681
26.2.4	P versus NP	682
26.3	Reductions	684
26.3.1	Median	685
26.3.2	Minimum and maximum	686
26.3.3	Interval scheduling	687
26.3.4	Reduction template	689
26.4	Problem hardness	690
26.4.1	Comparing problems	690
26.4.2	Transitivity	691
26.4.3	The NP-hard class	692
26.4.4	The NP-complete class	692
26.4.5	P versus NP	693
26.5	Theory and Practice	693
26.5.1	Theory	693
26.5.2	Practice	694
26.6	Summary	696
26.6.1	Reductions	696
26.6.2	Problem classes	696
26.6.3	Problems	697
27	Computability	699
27.1	Turing machine	699
27.1.1	Definition	700
27.1.2	Parity bit	701
27.1.3	Implementation	703
27.1.4	Checking passwords	708
27.2	The Church–Turing thesis	709
27.2.1	Computational models	709
27.2.2	Universal models	710

27.2.3	Length of string	711
27.3	Static analysis	714
27.3.1	Functions on functions	714
27.3.2	Writing functions on functions	716
27.3.3	Navel gazing	717
27.4	Undecidability	718
27.4.1	The halting problem	718
27.4.2	The totality problem	720
27.4.3	Rice's theorem	720
27.4.4	The equivalence problem	721
27.4.5	Reduction and computability	721
27.4.6	The problem landscape	722
27.5	Summary	723
27.5.1	Turing machines	723
27.5.2	Computability	724
28	TMA 03 part 2	727
28.1	Problem classes	727
28.1.1	Computable	728
28.1.2	Tractable	728
28.1.3	Intractable	729
28.1.4	NP-hard	729
28.1.5	NP	729
28.1.6	NP-complete	730
28.2	Turing machines	730
28.3	Python	731
28.3.1	Standard library	731
28.3.2	M269 library	731
28.4	Final words	731
29	Hints	733
29.1	Numbers and sequence	733
29.1.1	Expressions	733
29.1.2	Functions in mathematics	733
29.1.3	Functions in Python	733
29.1.4	Complexity	733
29.1.5	Run-times	734
29.2	Booleans and selection	734
29.2.1	Booleans	734
29.2.2	Decision problems	734
29.2.3	Boolean expressions	735
29.2.4	Classification problems	735
29.2.5	Practice	735
29.3	Sequences and iteration	736
29.3.1	The Sequence ADT	736
29.3.2	Strings	736
29.3.3	Iteration	737
29.3.4	Linear search	737

29.3.5	Tuples and tables	737
29.3.6	Lists	738
29.3.7	Reversal	738
29.3.8	Optional practice	739
29.4	Implementing sequences	739
29.4.1	Developing data types	739
29.4.2	Bounded sequences	740
29.4.3	Using dynamic arrays	740
29.4.4	Linked lists	740
29.5	Stacks and queues	740
29.5.1	Stacks	740
29.5.2	Using stacks	741
29.5.3	Queues	741
29.5.4	Priority queues	742
29.6	Unordered collections	743
29.6.1	Maps	743
29.6.2	Dictionaries	743
29.6.3	Hash tables	743
29.6.4	Sets	743
29.7	Practice 1	744
29.7.1	Pangram	744
29.7.2	Election	745
29.7.3	Voucher	745
29.7.4	Trains	746
29.7.5	SMS	746
29.8	Exhaustive search	747
29.8.1	Linear search, again	747
29.8.2	Factorisation	747
29.8.3	Constraint satisfaction	747
29.8.4	Searching permutations	747
29.8.5	Searching subsets	747
29.8.6	Practice	748
29.9	Recursion	748
29.9.1	Recursion on integers	748
29.9.2	Length of a sequence	749
29.9.3	Inspecting sequences	749
29.9.4	Creating sequences	749
29.9.5	Avoiding slicing	749
29.9.6	Multiple recursion	749
29.10	Divide and conquer	750
29.10.1	Variable decrease	750
29.10.2	Binary search	750
29.10.3	Binary search variants	750
29.11	Sorting	750
29.11.1	Insertion sort	750
29.11.2	Selection sort	750
29.11.3	Merge sort	750
29.11.4	Quicksort	751

29.11.5 Quicksort variants	751
29.12 Rooted trees	751
29.12.1 Algorithms on trees	751
29.12.2 Traversals	751
29.12.3 Binary search trees	751
29.12.4 Balanced trees	752
29.12.5 Heapsort	752
29.13 Graphs 1	752
29.13.1 Modelling with graphs	752
29.13.2 Basic concepts	753
29.13.3 Edge list representation	753
29.13.4 Adjacency matrix representation	753
29.13.5 Adjacency list representation	753
29.13.6 Classes for graphs	753
29.13.7 Traversing a graph	753
29.14 Greed	754
29.14.1 Minimum spanning tree	754
29.14.2 Shortest paths	754
29.15 Practice 2	754
29.15.1 Jousting	754
29.15.2 Dot product	755
29.15.3 Beams	755
29.15.4 Up and down	756
29.15.5 A knight goes places	756
29.16 Graphs 2	756
29.16.1 Undirected graph components	756
29.16.2 Directed graph components	757
29.16.3 Topological sort	757
29.16.4 State graphs	757
29.16.5 Practice	757
29.17 Backtracking	758
29.17.1 Back to the TSP	758
29.17.2 Back to the knapsack	758
29.18 Dynamic Programming	758
29.18.1 Longest common subsequence	758
29.18.2 Knapsack	759
29.19 Practice 3	759
29.19.1 Safe places	759
29.19.2 Extra staff	760
29.19.3 Borrow a book	760
29.19.4 Levenshtein distance	761
29.19.5 Higher and higher	761
29.20 Complexity classes	762
29.20.1 Tractable and intractable problems	762
29.20.2 The P and NP classes	763
29.20.3 Reductions	763
29.21 Computability	763
29.21.1 Turing machine	763

29.21.2 The Church–Turing thesis	764
29.21.3 Undecidability	764
30 Answers	765
30.1 Numbers and sequence	765
30.1.1 Arithmetic operations	765
30.1.2 Expressions	765
30.1.3 Assignments	765
30.1.4 Functions in mathematics	766
30.1.5 Functions in Python	766
30.1.6 Complexity	767
30.1.7 Run-times	767
30.2 Booleans and selection	769
30.2.1 Booleans	769
30.2.2 Decision problems	770
30.2.3 Boolean expressions	771
30.2.4 Classification problems	772
30.2.5 Practice	773
30.3 Sequences and iteration	778
30.3.1 The Sequence ADT	778
30.3.2 Strings	778
30.3.3 Iteration	780
30.3.4 Linear search	780
30.3.5 Tuples and tables	782
30.3.6 Lists	783
30.3.7 Reversal	784
30.4 Implementing sequences	787
30.4.1 Developing data types	787
30.4.2 Bounded sequences	788
30.4.3 Linked lists	788
30.5 Stacks and queues	789
30.5.1 Stacks	789
30.5.2 Using stacks	789
30.5.3 Queues	792
30.5.4 Priority queues	794
30.6 Unordered collections	797
30.6.1 Maps	797
30.6.2 Dictionaries	797
30.6.3 Hash tables	798
30.6.4 Sets	799
30.7 Practice 1	800
30.7.1 Pangram	800
30.7.2 Election	803
30.7.3 Voucher	804
30.7.4 Trains	805
30.7.5 SMS	808
30.8 Exhaustive search	810
30.8.1 Linear search, again	810

30.8.2	Factorisation	811
30.8.3	Constraint satisfaction	812
30.8.4	Searching permutations	812
30.8.5	Searching subsets	812
30.8.6	Practice	813
30.9	Recursion	815
30.9.1	Recursion on integers	815
30.9.2	Length of a sequence	816
30.9.3	Inspecting sequences	816
30.9.4	Creating sequences	818
30.9.5	Avoiding slicing	819
30.9.6	Multiple recursion	819
30.10	Divide and conquer	821
30.10.1	Variable decrease	821
30.10.2	Binary search	821
30.10.3	Binary search variants	821
30.11	Sorting	823
30.11.1	Insertion sort	823
30.11.2	Selection sort	823
30.11.3	Merge sort	824
30.11.4	Quicksort	824
30.11.5	Quicksort variants	824
30.12	Rooted trees	825
30.12.1	Binary tree	825
30.12.2	Algorithms on trees	825
30.12.3	Traversals	826
30.12.4	Binary search trees	826
30.12.5	Balanced trees	827
30.12.6	Heapsort	829
30.13	Graphs 1	829
30.13.1	Modelling with graphs	829
30.13.2	Basic concepts	830
30.13.3	Edge list representation	831
30.13.4	Adjacency matrix representation	832
30.13.5	Adjacency list representation	832
30.13.6	Classes for graphs	833
30.13.7	Traversing a graph	833
30.14	Greed	834
30.14.1	Interval scheduling	834
30.14.2	Minimum spanning tree	834
30.14.3	Shortest paths	836
30.15	Practice 2	837
30.15.1	Jousting	837
30.15.2	Dot product	838
30.15.3	Beams	840
30.15.4	Up and down	843
30.15.5	A knight goes places	845
30.16	Graphs 2	848

30.16.1 Undirected graph components	848
30.16.2 Directed graph components	849
30.16.3 Topological sort	849
30.16.4 State graphs	850
30.16.5 Practice	851
30.17 Backtracking	851
30.17.1 Back to the TSP	851
30.17.2 Back to the knapsack	853
30.18 Dynamic Programming	853
30.18.1 Longest common subsequence	853
30.18.2 Knapsack	856
30.19 Practice 3	859
30.19.1 Safe places	859
30.19.2 Extra staff	860
30.19.3 Borrow a book	861
30.19.4 Levenshtein distance	862
30.19.5 Higher and higher	866
30.20 Complexity classes	869
30.20.1 Tractable and intractable problems	869
30.20.2 The P and NP classes	870
30.20.3 Reductions	870
30.21 Computability	871
30.21.1 Turing machine	871
30.21.2 The Church–Turing thesis	872
30.21.3 Undecidability	873

31 Figures	875
31.1 Sequences and iteration	875
31.1.1 Reversal	875
31.2 Implementing sequences	875
31.2.1 Static arrays	875
31.2.2 Bounded sequences	877
31.2.3 Linked lists	877
31.3 Stacks and queues	878
31.3.1 Stacks	878
31.3.2 Queues	878
31.4 Unordered collections	878
31.4.1 Hash tables	878
31.5 Practice 1	879
31.5.1 Trains	879
31.6 Exhaustive search	879
31.6.1 Searching permutations	879
31.7 Sorting	880
31.7.1 Merge sort	880
31.7.2 Quicksort	880
31.8 Rooted trees	880
31.8.1 Binary tree	880
31.8.2 Traversals	881

31.8.3	Binary search trees	881
31.8.4	Balanced trees	882
31.8.5	Heapsort	882
31.9	Graphs 1	883
31.9.1	Modelling with graphs	883
31.9.2	Basic concepts	884
31.9.3	Edge list representation	885
31.9.4	Adjacency matrix representation	885
31.9.5	Adjacency list representation	885
31.10	Greed	885
31.10.1	Interval scheduling	885
31.10.2	Weighted graphs	886
31.10.3	Minimum spanning tree	886
31.10.4	Shortest paths	887
31.11	Practice 2	887
31.11.1	Beams	887
31.11.2	Up and down	888
31.11.3	A knight goes places	888
31.12	Graphs 2	888
31.12.1	Undirected graph components	888
31.12.2	Directed graph components	889
31.12.3	Topological sort	889
31.12.4	State graphs	890
31.13	Backtracking	891
31.13.1	Generate sequences	891
31.13.2	Trackword	891
31.13.3	Back to the TSP	891
31.13.4	Generate subsets	891
31.14	Dynamic Programming	892
31.14.1	Fibonacci	892
31.14.2	Longest common subsequence	892
31.14.3	Knapsack	893
31.15	Practice 3	893
31.15.1	Extra staff	893
31.15.2	Borrow a book	894
31.15.3	Levenshtein distance	894
31.15.4	Higher and higher	894
31.16	Complexity classes	895
31.16.1	The P and NP classes	895
31.16.2	Reductions	895
31.16.3	Problem hardness	895
31.16.4	Summary	896
31.17	Computability	897
31.17.1	Turing machine	897
31.17.2	The Church–Turing thesis	897
31.17.3	Undecidability	897
31.17.4	Summary	898

To Clara, Ana and Claudia

This textbook is part of the 2025/26 presentation of the Open University module M269 *Algorithms, data structures and computability*.

This is the version as of 29 August 2025.

Copyright © 2020–2025 The Open University

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd, 5th Floor, Shackleton House, 4 Battle Bridge Lane, London SE1 2HX (website www.cla.co.uk).

Open University materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Acknowledgements

I thank my colleagues Alistair Willis, Andy Connolly, Phil Molyneux, Shena Deuchars (critical readers), Jonathan Darch (editor) and Andrew Whitehead (graphic designer) for their input, which greatly improved the content and presentation of this book. Thanks are also due to Alan Bradshaw, Allan Thrower and Mark Hall for specifying, procuring and configuring the JupyterHub server.

Many thanks to Tony Hirst and all students and tutors who reported typos and other mistakes.

Thanks to Margarida Mamede for suggesting some exercises and to Steve Halim for his [Methods to Solve](#) site, from which I selected some problems.

A very special thanks to the rest of the M269 team: Adam Linson, Alexis Lansbury, Ann Walshe, Brendan Quinn, Jane Evans, Oli Howson and Phil Hackett. Without their support and their hard work to get the myriad other things done (write TMAs and marking guides, hire tutors, produce all the paperwork and budgets necessary, etc. etc.) the new M269 wouldn't exist. Last but not least, many thanks to the tutors for their tutorials, student support, marking feedback and for coping with the many changes introduced by the new edition.

CHAPTER 1

INTRODUCTION

Money may make the world go round, but it's algorithms that make money go round. When you shop online or in-store, algorithms are the step-by-step procedures that computers follow to apply discounts and taxes, to encrypt and transmit your card details, to do any currency conversions, and to finally debit your account and credit the seller's.

Algorithms recommend what to read, watch and buy, whom to date and whether we should get a loan. Algorithms predict the weather, drive cars, recognise faces and trade in the stock market. Our lives increasingly depend on algorithms.

M269 is an introduction to what algorithms are and how they work. You will learn some general algorithmic techniques and ways of organising data so that algorithms can use it. Techniques, algorithms and data structures that are quite advanced or specific to particular domains (imaging, artificial intelligence, etc.) are outside the scope of this module.

Algorithms should produce their results as quickly as possible. You will learn how to measure the efficiency of an algorithm in a way that doesn't depend on which computer executes it. In the last part of this module you will become aware of the limitations of algorithms and learn about problems for which no efficient algorithm is currently known, and problems that can't be solved by any algorithm.

This chapter tells you what to expect in M269 and guides you in preparing for it.

1.1 What to expect

This section briefly describes what you can expect from M269 and what is expected of you.

1.1.1 Content

Algorithms, data structures and computability have been core topics of Computing courses around the world for decades, and for a good reason. The first two topics are eminently practical: knowledge of how to structure data, construct algorithms and analyse their efficiency is useful for the design and implementation of any software system. It's knowledge you can immediately put to use in other modules and in your professional life. The third topic is somewhat theoretical, but

it is *the* Computer Science topic. Computability addresses the fundamental questions of what computation is and what can and can't be computed. I'm sure you will find M269 useful and intellectually stimulating.

M269 covers most of the core algorithm, data structure and computability topics recommended for undergraduate computing curricula by professional bodies. The first third of M269 (Chapters 2–10) explains the basic data structures and programming constructs needed for the rest of M269. Those chapters are therefore the most intensive with respect to the amount of terminology and concepts introduced, but you should know several of them from prior study. Half of M269 (Chapters 11–25) is dedicated to algorithmic techniques, introducing further data structures, as needed. The final Chapters 26–28 are about the complexity and computability of computational problems. As in most modules, you can expect the material to get more difficult as you progress.



Info: The recommended core topics are listed [here](#).

A recurring theme of M269 is abstraction. As you shall see later, an algorithm is an abstraction of the steps executed by a computer, data structures abstract the relationships between the data being processed, and the complexity of an algorithm is an abstraction of its run-time.

Abstracting from problem-specific details leads to general models, rules and concepts that help us tackle new problems: that's the power of abstraction. The cover of this book illustrates the abstraction process, going from a detailed representation of a cow to a general geometric shape, the rectangle.



Info: The cover shows Theo van Doesburg's [Composition VIII \(The Cow\)](#) and three studies leading to it, all from 1917–18. MoMA's website has more [drawings](#) and more information on the [final study](#).

The emphasis of M269 is computational problem solving. The module gives you techniques and a process for solving several types of problems, comparing alternative solutions, and communicating problems and solutions clearly. Many examples illustrate all of this. Much of the teaching is done through examples and exercise solutions. Study them carefully. Don't just glance through them: make sure you understand each example and solution before moving on.

However, learning is not passive: it's an activity. We don't learn how to walk, drive a car or play a musical instrument just by watching others doing it. The only way to learn how to solve problems is to solve problems, and M269 is full of exercises for you to practise. Expect to spend most of your time figuring out how to solve a problem, trying out different approaches, analysing which one is best, writing, testing and debugging code, and documenting your solution.

Algorithms can be described in plain English but must be implemented in some programming language for computers to execute them. M269 uses Python because it's an expressive, readable and uncluttered programming language.

M269 aims for you to develop sought-after cognitive and professional skills around problem

solving, programming and communication. The intended learning outcomes of M269 are:

- Knowledge and understanding:
 - Understand the common general-purpose data structures, algorithmic techniques and complexity classes.
 - Know about the limits of computation and their practical implications.
 - Understand the Turing Machine model of computation.
- Cognitive skills:
 - Develop and apply algorithms and data structures to solve computational problems.
 - Analyse the complexity of algorithms to support software design choices.
- Key skills:
 - Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders.
- Practical and professional skills:
 - Write readable, tested, documented and efficient Python code.

1.1.2 Challenges

There's usually a jump in pace and difficulty from Stage 1 to Stage 2 modules and M269 is among the most challenging Stage 2 Computing modules. If you don't have experience of high-intensity study, in which you need to understand and apply many terms, concepts and techniques, you may wish to pass a Stage 2 module before starting M269.

M269 is challenging mainly for three reasons. First, it contains lots of material and over 5,000 lines of Python code in order to cover the recommended core topics in a single module, due to how the OU curriculum is organised. Other universities spread the M269 material (and additional topics) across two or three modules.

To cope with the amount of M269 material, previous students found it useful to:

- use [concept-mapping](#) software to organise the M269 topics and terminology
- look at the TMAs *before* working through the corresponding chapters to gauge what to study in depth, what to skim and what to skip. The TMAs are in the '[Assessment](#)' tab of the M269 website.

Second, M269 is about solving computational problems, which is inherently difficult: it requires experience, technical knowledge, persistence and creativity. No wonder good problem-solvers are sought after and well paid!

I've written parts of this book in a 'think aloud' style, so that you can see how I approach a problem. Another feature of this textbook is the question checklists in Chapters 5, 10, 15, 20, 25 and 28 to help steer you in the right direction when solving a problem.

Nevertheless, learning by yourself how to solve problems is difficult. Problem solving is best done together with others, to bounce off ideas and check potential solutions for errors and possible improvements. I encourage you to actively participate in the forums and tutorials. Just

reading other's posts or watching recorded tutorials doesn't provide the same learning experience as interacting with peers and tutors.

Third, the M269 topics are inherently abstract and not easy to convey just with text and static figures. Understanding how code works and how it changes data in memory is not easy from just reading the code. Some students found [Python tutor](#) useful to visualise the step-by-step execution of M269 code.

Let's not beat about the bush: M269 is hard work and challenging. But if you put in the effort, you'll gain conceptual and practical problem-solving skills sought by employers.

1.1.3 Previous knowledge

I assume you have had an introduction to Python, like in TM112, and that you can write simple short programs using the following constructs:

- variables, expressions and assignments
- integers, Booleans, strings, lists and some operations on them
- selection statements (if-elif-else and nested ifs)
- iteration statements (while-loop, for-in-range, for-in-collection)
- function definitions and calls.

Chapters 2 to 4 recap these constructs, but you may find the pace too fast and the explanations too brief if you don't have Python experience. The initial chapters include new material too, so don't skip them even if you are familiar with Python.

I also assume you have basic mathematical proficiency:

- you understand and know how to use percentages, powers (like n^4 and 2^n), logarithms and scientific notation (like 4.3×10^{-8})
- you can write and evaluate expressions for 'word problems' like calculating the service charge as a percentage of a restaurant bill
- you can manipulate algebraic expressions, e.g. transform $x + 4y = 20$ into $y = 5 - x/4$.

You can check your Python and maths knowledge with the 'Are you ready for M269' quiz, if you haven't done it before registering for M269. The quiz and revision resources are available from the [M269 page](#) of the Computing & IT study site.

Activities

Before continuing with the next section, do the [Getting Started](#) activities for Section 1.1, to help you get a better idea of what to expect in M269. This includes watching a BBC programme on algorithms; some of them will be presented in more detail in M269.

1.2 How to study

To successfully complete M269, see it as a project: it has a start and an end, goals (the learning outcomes), a schedule (the study planner) and deliverables (the TMAs). Like any project, M269 requires planning and preparation. The first step is to get to know the M269 materials, which consist of this book, three TMAs, and other resources on the [M269 website](#).

1.2.1 Book

This book is provided in three formats: Jupyter notebooks, PDF and HTML. Each has its advantages.

The PDF format is read-only. Using a free PDF reader, you can read the PDF book on a tablet, use the PDF's table of contents to jump to any subsection, and quickly search for a particular word over the whole book. Depending on the PDF reader and the accessibility settings of your computer, you can even have the book's content read to you, e.g. during commuting, but code and technical notation may be read out incorrectly. A PDF reader also allows you to bookmark pages, highlight passages, add notes and print pages, e.g. the summary section at the end of most chapters.

The HTML book is read-only too. You can read it with any web browser. As with the PDF book, you can search for any word, go directly to any section, and possibly have the content read to you. You can also bookmark and print HTML pages. Each HTML page is a complete section, corresponding to several PDF pages. You can change the font size of the HTML book with your browser's zoom in and zoom out commands.

Jupyter notebooks are interactive documents that contain both text and code. I mean real, executable, editable code, not static text in a monospaced font. With notebooks, you don't need to constantly switch between a text-only book and a programming environment. So you'll have more time for practice. The M269 TMAs are notebooks too, to save you and your tutor time from handling separate files with the TMA questions, your answers and your code.

Notebooks are editable. You will edit the notebooks we provide to fix errata and to write answers to exercises and TMA questions. You may add study notes, alternative solutions, wrong solutions and comments why they're wrong. Upon completing M269, you will have unique notebooks, with *your* answers and notes.

Working with Jupyter notebooks requires a laptop or desktop computer, with a proper keyboard and a decent-sized screen. There's one Jupyter notebook per book section, to keep them relatively small. Unlike with the PDF and HTML formats, you can't easily search for a term across all notebooks, only within a single notebook. It's also not as easy to highlight passages and make notes in Jupyter notebooks.

Notebook files have the extension '.ipynb', which comes from their original name: IPython notebooks. (IPython stands for interactive Python.) Do *not* double-click on a notebook to open it: this will probably open a text editor showing the internal format of a notebook. If you have double-clicked on a notebook, then immediately close the editor without changing the file. The next section explains how to install the necessary software to use notebooks. Once the software is installed, you'll be able to work on notebooks with a web browser.

If for accessibility or other reasons you can't or don't want to use Jupyter notebooks, you can

instead use your preferred code editor with the provided Python files, one per notebook, with all the code in that notebook. Some code is specific to Jupyter, but most of it can be executed in any Python environment.

Most book sections have exercises. At the end of each one there's a link to its answer. Most exercises also have a link to a hint. Each hint and answer is in a separate notebook (and separate page in the HTML version), so that you don't accidentally see the hint or answer for the next exercise. However, the PDF version has several hints and answers on the same page. Once you have read the hint or answer in HTML or PDF, you can return to the exercise by using the 'back' command of your browser or PDF app.

This book uses a few conventions, explained next. All formats look slightly different, due to the way the PDF and HTML books are generated from the notebooks.

Exercises are numbered consecutively from 1 in each section, so that you can easily refer to them in the forums and tutorials. For example, Exercise 1.3.2 is the second exercise of Section 3 of Chapter 1.

Besides exercises, the book has short questions to check your understanding of the concepts. Each question and its answer are separated by a horizontal line like this:

Consider the lines as 'stop and think' signs. At these points you should stop reading, answer the question in your head, and then continue.

Terms in bold, e.g. **algorithm**, may occur in the assessment. You must be able to understand these terms and use them in your answers. You don't have to memorise definitions.



Info: I use information boxes like this one to indicate alternative terms you may find in other books or on the web, to refer back to TM112, to provide the sources of exercises and data, etc. You can skip information boxes: they won't be assessed. Information boxes look differently across book formats, but always start with 'Info'.



Note: I use advice boxes like this one for important teaching points, e.g. to help you solve problems and implement algorithms more effectively. Most advice boxes are in the main sections, but some appear in the solutions to exercises. Advice boxes look differently across book formats, but always start with 'Note'.

Code examples look like this:

```
[1]: def list_length(a_list):
    """Return the length of a list."""
    length = 0
    for item in a_list:
```

(continues on next page)

(continued from previous page)

```
length = length + 1
return length
```



Info: This function appears in TM112 Block 2 Section 4.1.4.

If the code produces some result, it's shown immediately below:

```
[2]: list_length(["on your marks", "get set", "go!"])
[2]: 3
```

Note the use of **syntax colouring** to highlight the different parts of the code: strings, special words, function names, etc. Each book format may use a different colour scheme.

Code fragments and templates, i.e. text that looks like code but can't be executed, is typeset as follows.

```
for item in collection:
    process the item
```

Code within a paragraph of text is typeset like this: `length = length + 1`.

Occasionally, code lines are longer than the page width, like this one:

```
[3]: a_long_variable_name_for_a_long_and_silly_string =
    ↪'supercalifragilisticexpialidocious is quite atrocious rather than_
     ↪delicious'
```

In the PDF book, long lines are split in two or more. Each line besides the first one starts with the symbol ↪ to show it's the continuation of the previous line. Spaces at the end of a line are explicitly shown with the symbol ↵. In the Jupyter and HTML books, use the mouse to scroll the long line to the right and read the rest of it.

1.2.2 Mindset

An important part of your study preparation is to get into the right mindset.

As I mentioned, problem solving requires practice: lots of it, and regularly. There's no recipe for solving computational problems, especially when we're looking for the most efficient algorithm. There are general techniques, but it requires creativity to select the right one and adapt it to the problem at hand. You must view the problem from different perspectives, explore alternatives, weigh their pros and cons. Creativity is in part a systematic exploration of different approaches. There will be dead ends and you need to give your brain a break to start afresh. It's hard to have marathons of creativity. Think little and often about the M269 exercises. Use idle periods (like queuing) to mull over them, preferably every day.

As toddlers we stood up on our own, tried to walk, fell and got hurt. That didn't stop us continuing trying and eventually succeeding. Likewise, when solving problems our initial attempts probably won't work: our algorithms will not compute the correct output for some inputs, they won't be efficient for large inputs, the code will crash. At such times, it's easy to get frustrated and fed up, give up, and move to the next exercise. It's easy and wrong. We learn more from our mistakes than from our successes. Throughout the book I'll give you tips on how to handle errors, but it helps from the outset if you take mistakes in your stride, as an unavoidable part of problem solving, instead of as something to despair about.

You should also remember that you're not alone and that problem solving is a social activity. In professional practice, problems are solved in teams that bounce ideas around, discuss alternative designs, and look at each other's code. You can discuss the exercises with your tutor and in the forums, ask for help, and even work on the exercises together with other students. Studying part-time at a distance can feel lonely at times, so you may wish to have some 'study buddies' whom you meet regularly online or face to face.

Note however that you'll have to work on the assignments by yourself. Employers need assurances that you have the knowledge and the skills to become an active player in their problem-solving teams. You can however ask for clarifications: if a question isn't clear, contact your tutor.



Note: Problem solving requires regular hard work, grit, resilience, and a willingness to fail and to ask for help.

1.2.3 Time allocation

M269 is a 30-credit module over 30 weeks, so expect to work for an average of 10 hours per week. If you're studying part-time, I strongly advise you to do at most one other 30-credit module in parallel to M269.

Each week usually corresponds to one book chapter. The material gets progressively harder, so early weeks like this one don't require 10 hours to read and work through, while others may require more.



Note: Get ahead of the study planner whenever you can.

Don't forget to plan for activities done outside the book, like installing software, participating in tutorials and forums, and working on the TMAs. This book contains several optional exercises. They are not accounted for in the workload.

The study planner allocates two study-free weeks to each TMA. Use them well. The corresponding book chapters don't introduce new material: they just provide some guidance for the TMA. If you're studying more than one module, check all the submission dates and plan your work on TMAs accordingly.

As part of your planning, set up study periods that fit your personal and work life pattern. Don't fret if they get occasionally thrown off by life events: the TMA weeks give you some slack to catch up. As you study M269 and gain experience in what works best for you, be prepared to change your initial plan.

M269 requires (and aims to develop) two important transferable skills: coping with lots of technical information in a short period of time, and being pragmatic about what can be done to meet a deadline.

1.2.4 Study techniques

Psychology studies on how we learn have made some surprising findings over the decades. Some of the learning techniques that studies found to be effective are:

1. Vary your study environment, so that the brain gets different stimuli.
2. Self-test regularly, e.g. with flashcards or by explaining what you learned to others.
3. Sleep, to help the brain re-process information and build connections.
4. If you're stuck with a problem, take a break and do something unrelated.
5. Break up study time, to force the brain to dig up again the information and re-store it.
6. Start a longer project as early as possible, to help the brain be attuned to information that is relevant to the project.

Technique 1 is not just about varying the time and place of study, but also the way you study: reading, taking notes, discussing with peers, etc. However, keeping re-reading and highlighting the same material, or rewriting your notes, is a very passive way of learning, in which your brain is not recalling or applying the material learned. Hence the need for technique 2.

Every year some M269 students post in the forums that something they were struggling with suddenly 'clicked' after dinner or the next day, once they looked at it with 'fresh eyes'. That's techniques 3 and 4 at work. Note that working on a problem for 2–3 minutes doesn't count as 'being stuck'. The brain needs to grapple with a problem or a concept for some time and from different angles to 'absorb' it. There's of course no guarantee that 'sleeping on it' will reveal a solution to you. Rather, not giving enough rest to the brain will make learning harder.

A recurring theme emerges from the above: the brain seems to learn best when it is repeatedly engaging with the same material but in different ways and when it is given the time to process the acquired material. The brain needs to forget and be probed again to learn what is important to remember.



Info: For more details on these techniques and the studies behind them, read Benedict Carey's *How we learn*.

Studying better is not about putting in more hours, but about using the time you have available more wisely. Here are some suggestions for M269, repeating some given previously.

- Don't study 5h on one day and five on the next, for example. It's better to split the 10 weekly hours across three or four non-consecutive days. (Technique 5)
- Instead of listening to a tutorial recording, it's better to spend the same time attending it and interacting with tutor and peers. (1)
- Interleave reading and coding: that's why M269 is delivered as Jupyter notebooks. (1)
- If you don't understand something after the second reading, go out of the 'book environment': ask your tutor, ask on the forum, attend a tutorial, do a concept map, use visualisations and other third-party material, etc. (1)
- Do flashcards with the terms in the summary sections of each chapter. (2)
- Attempt to answer other students' queries on the forums, even if you're not very sure of your answer. (2)
- Don't skip the stop-and-think lines or the exercises; try to do the optional ones. (2)
- Attempt the exercises in chapters 9, 19 and 24 because they draw on material in earlier chapters, thus making you revisit concepts and apply them. (2 and 5)
- Before working through a chapter, read (or skim again) the next TMA, to be aware of which material is most relevant. (6)

Activity

Now do the [Getting Started](#) activities for Section 1.2.

1.3 Software

As mentioned before, this book and the TNAs are in the form of Jupyter notebooks so that you can run and modify the code we provide, and write your own code for exercises and assessment.

You can use any platform that supports Jupyter notebooks, but the one your tutor and the module team support is the JupyterLab interface in a web browser.



Note: To avoid problems later on, update your browser now to its most recent version.



Info: If you're familiar with an integrated development environment that supports notebooks, like Visual Studio Code and PyCharm, you may prefer to use it instead of JupyterLab.

You have two options for working with JupyterLab. You can use the M269 virtual computing environment (VCE) on the OU's Open Computing Lab, or you can install JupyterLab locally on your desktop or laptop.

Using the VCE has two advantages: you can work on a tablet (but it may not be very practical) and you don't have to install or configure any software. However, if there's no internet connection or there's a problem with the OU servers, you won't be able to access the VCE. Using a local installation on your desktop or laptop allows you to work offline, but you must install and configure the M269 software.

You can use both the VCE and a local installation, but it can get tedious. As you work through the notebooks and modify them, you need to upload and download them to and from the VCE to keep your local and remote workspaces in sync. To avoid that, I suggest you use the VCE and the local installation for different purposes, e.g. use one to just read the notebooks and the other to do the exercises.



Note: If you modify the VCE notebooks, set a recurring reminder to back up them up.

The easiest way to back up your VCE content is to create a zip archive on the VCE and then download that file to your computer.

To work through this book on your local machine, follow these steps.

1. Create a folder named `m269-25j` or `M269-25J` somewhere on your disk. This folder will be referred to as your M269 folder. It can be put in a cloud drive, like Google Drive or OneDrive, if you want to access it from multiple computers and automatically back up any changes.
2. Download the M269 book zip archive from the [Resources](#) tab of the M269 website (not from the VCE!) to your M269 folder and unzip it there.
3. Install the M269 software and start JupyterLab, as explained in <https://dsa-ou.github.io/m269-installer>.

Whether you are using the VCE or the locally installed software, the rest of this week is for you to learn how to use JupyterLab. See the [Getting Started](#) pages of the M269 website.

1.4 Summary

The following summarises the main points about using notebooks in JupyterLab. Other Jupyter platforms behave slightly differently. You may wish to print this notebook or keep it open until you're familiar with the commands.

I follow the convention of writing letter keys as they're printed on keyboards: in uppercase. `Ctrl-A` means to just press those two keys and is different from `Ctrl-Shift-A`. Likewise, `A` and `Shift-A` are different commands.

1.4.1 Notebooks

A notebook is a sequence of text and code cells. Each notebook is associated to a kernel (a Python interpreter) that executes the code cells of that notebook and holds the variables' state. Running a text cell formats its text; running a code cell executes its code.

At any point, the notebook is either in command or edit mode. Command mode allows us to change whole cells or the notebook, while edit mode allows us to change the content of the currently selected cell, which is indicated by a thick bar to its left. In edit mode, you can see a cursor inside the current cell.

To switch to edit mode, press Enter, click inside a code cell, or double-click inside a text cell. To switch to command mode, press the Escape key or click to the left of a cell.

To use notebooks in a productive way:

- Prefer using the keyboard to the mouse.
- Learn the common shortcuts to copy/cut/paste text, undo/redo edits, move the cursor to the start/end of the word/line, select text with the keyboard, etc.
- Create a checkpoint when you open the notebook and after completing each exercise.
- Press Ctrl-Shift-Q to close and halt a notebook (stop its kernel). Don't just close the notebook's tab.

To halt all open notebooks and exit JupyterLab, select File > Shut Down from the menu bar.

1.4.2 Common commands

The following can be used in both edit mode and command mode. (If you're reading this section in HTML, you may have to scroll the next two tables to the right to see all their columns.)

Effect	Key	Menu bar	Tool bar
save notebook, create checkpoint	Ctrl-S or Cmd-S	File > Save and Checkpoint	disk icon
run current cell, switch to command mode	Ctrl-Enter or Cmd-Enter	Run > Run Selected Cell and Do not Advance	
run current cell, switch to command mode, select next cell	Shift-Enter	Run > Run Selected Cell	play
replace text in the current cell or all cells	Ctrl-F or Cmd-F	Edit > Find	
list of keystrokes		Help > Show Keyboard Shortcuts	
run first to the last cell, stop at first error		Run > Run All Cells	
shutdown kernel, start new one, then as above		Run > Restart Kernel and Run All Cells	fast forward
run first to previous cell, stop at first error		Run > Run All Above Selected Cell	
discard changes, reinstate last checkpoint		File > Revert Notebook to Checkpoint	
close notebook, shut down kernel	Ctr-Shift-Q	File > Close and Shut Down Notebook	

1.4.3 Command mode

The following keyboard shortcuts can be used in command mode only.

Effect	Key	Menu bar	Tool bar
select previous/next cell	up/down arrow		
add code cell above current cell, select it	A		
add code cell below current cell, select it	B		plus icon
copy current cell	C	Edit > Copy Cell	copy icon
paste copied cell below current cell	V	Edit > Paste Cell Below	clipboard icon
delete current cell	DD	Edit > Delete Cell	
change current cell to a (Markdown) text cell	M		dropdown
change current cell to a (pYthon) code cell	Y		dropdown
show/hide line numbers in all code cells	Shift-L		
scroll the notebook down/up	Space / Shift-Space		
switch to edit mode	Enter		

1.4.4 Edit mode

The following apply to text and code cells.

Effect	Key
indent (move right) current line (or selected lines)	Ctrl-] or Cmd-]
dedent (move left) current line (or selected lines)	Ctrl-[or Cmd-[
switch to command mode, don't run current cell	Esc
comment/uncomment current line (or selected lines)	Ctrl-/ or Cmd-/

The following only applies to code cells.

Effect	Key
autocomplete the word to the left of the cursor	Tab

CHAPTER 2

NUMBERS AND SEQUENCE

An algorithm is a step-by-step procedure that takes some inputs and, on completion, produces some outputs. For example, a recipe is an algorithm for transforming ingredients into a dish. A flatpack assembly manual provides an algorithm to put wood, nuts, bolts, etc. together into a piece of furniture. Satnav directions are an algorithm to take your car from its current location to the desired location.

Algorithms should be correct, i.e. lead to the desired output, and usually we want them to be efficient too, i.e. lead to the output as quickly as possible. Sometimes we prefer a slower algorithm, for example because it leads to a tastier dish or takes the scenic route.

In M269 we are interested in algorithms that can be carried out by computers. This has two implications. First, the inputs and outputs are data instead of physical objects. Second, the instructions must be unambiguously stated, because computers have no capacity for judgement. For example, recipe instructions like ‘season to taste’ or ‘fry until golden brown’ are not precise: different people will interpret them differently. For the purposes of M269, an **algorithm** is a step-by-step procedure that is expressed as a structured list of unambiguous instructions.

This and the next two chapters recap the basic kinds of data and instructions used in computational algorithms. This chapter in particular covers numeric data and arithmetic instructions, structured as a **sequence** of steps to be executed one after the other. This chapter also defines the concepts of correctness and efficiency for computational algorithms.

We will express algorithms in English (with some mathematical notation as needed) and in Python, but M269 is not a Python course. The emphasis is on understanding, creating and analysing algorithms; Python is just a means to execute and test the algorithms. I will introduce as little Python as I can get away with: just enough to express in a clear way the algorithms and data structures M269 covers. The advantage of restricting the Python features we use is that it becomes easier to translate an algorithm to Python. The disadvantage is that our code won’t be idiomatic, i.e. it won’t use the Python language in the best way.

This chapter supports the following learning outcomes.

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter covers numeric data and sequential algorithms, and

introduces algorithmic complexity.

- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – this chapter introduces preconditions to describe assumptions on the input values.
- Write readable, tested, documented and efficient Python code – this chapter introduces Python coding conventions and documentation strings.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

2.1 Numbers

A **data type** is a collection of values and operations on those values. For example, the integer data type includes values like -1, 0 and 1, and operations like addition and multiplication. When referring to a type independently of how it's implemented, we'll use the term **abstract data type**, abbreviated **ADT**. ADTs are usually named after their values, in the singular.

This and the next section introduce the numeric types used in M269: the integer ADT, the real number ADT, and the corresponding Python data types. This section covers the values; the next section covers the operations.

2.1.1 Integers and real numbers

The **real numbers** are the numbers that can be represented in decimal form, with a finite or infinite number of digits after the decimal point. The real numbers include the **integers** (0, 1, -1, 2, -2, etc.) and numbers like $\frac{1}{3} = 0.333\dots$ and $\pi = 3.141592\dots$



Info: The integers are also called integer numbers or whole numbers.



Note: In M269 we won't use the alternative terms mentioned in the information boxes.

The **positive** numbers are those greater than zero; the **negative** numbers are those less than zero. The **natural numbers** are the non-negative integers: 0, 1, 2, etc. They are typically used for counting, e.g. how many copies of a certain product are in stock: if a product is out of stock, its count is zero.



Info: Some texts, like the MU123 and MST124 books, don't consider zero to be a natural number. Computing texts usually do, because natural numbers are also used for indexing, as we shall see in Chapter 4.

Many real numbers can be represented in various ways, e.g. 0.5 can be represented as a fraction ($\frac{1}{2}$), as a percentage (50%) and in scientific notation (5×10^{-1}). The latter representation is useful for very large and very small numbers. For readability, groups of three digits are separated with commas, e.g. 9,500,000 is nine and a half millions.



Info: MU123 Units 1 and 3 and MST124 Unit 1 introduce the various kinds of numbers and notations.

Numbers are often accompanied by units of measurement. Depending on the unit and the required precision, we may use integers or real numbers. For example, we can represent a person's height as 1.7 metres or 170 centimetres. As another example, travel sites indicate temperature values with integers, but engineering applications need more precision and use real numbers.

2.1.2 int and float literals

Python provides several numeric data types. M269 uses types `int` and `float`.

Python's `int` type is, for practical purposes, indistinguishable from the integer ADT, because Python can represent any integer, no matter how many digits it has, provided there's enough memory.

A **literal** is a direct representation of a value. In Python, an integer literal is one or more digits, like 9500. Note that in Python, numeric literals don't include a minus sign. I'll come back to this point later.

Python's `float` type only includes a restricted subset of the real numbers, namely those that can be represented as a binary **floating-point number** (`float` for short) with some fixed number of binary digits. (The exact number of digits available may depend on the Python interpreter and the hardware used.)

In Python, a number written with a decimal point is a float literal (even if it represents an integer number), otherwise it's an integer literal.

```
[1]: 2.0 # a float literal
```

```
[1]: 2.0
```

```
[2]: 2 # an integer literal
```

```
[2]: 2
```

The text after the hash symbol `#` is a **comment**. Comments are ignored by the Python interpreter; they are notes for us, human readers of the code.

The real number π occurs in many mathematical formulas. Python provides a float with an approximate value. Python's definition of π is in a separate **module** (code library) named `math`, because not all applications need it. To use a module, we must tell the interpreter to **import** it, like so:

```
[3]: import math
```

This line of code generates no output, but the number to the left of the cell is a confirmation that it has been executed. The syntax colouring shows that the two words have a different ‘status’: `import` is a Python **keyword**, a word with a special meaning, while `math` is an **identifier**, simply a name.

Now we can access the approximate value of π like so (note the dot):

```
[4]: math.pi
```

```
[4]: 3.141592653589793
```

Precise calculations with reals may become imprecise with floats due to their internal binary representation. For example, the product of 0.1 and 3 is not 0.3 when using floats, because there’s no binary floating-point number that represents exactly 0.1.

```
[5]: 0.1 * 3
```

```
[5]: 0.3000000000000004
```



Info: TM112 Block 1 Section 1.2 explains floating-point numbers and their binary representation.

M269 rarely uses floats; they occur mainly when measuring the run-time of code.

2.1.3 Mistakes

Programming language interpreters are either very picky or very lax about the form in which they expect instructions to be. Deviations from the expected form can lead to error messages or subtle mistakes that may be difficult to detect. This and similar subsections throughout the book show you some typical mistakes when writing code and how the interpreter reacts to them.

Python uses commas to separate items, as we shall see in Chapter 4. So, if you use a comma as a thousands separator or as a decimal point, then the interpreter will consider the input to be several separate numbers.

```
[6]: 70,500 # this is two integers, not seventy and a half thousand
```

```
[6]: (70, 500)
```

```
[7]: 3,42 # wrong decimal separator: must be .
```

```
[7]: (3, 42)
```

This mistake is hard to detect, because it’s valid Python syntax. If you get an error message about the interpreter expecting a number instead of a list of numbers, then the source of the error might be this one.

If the interpreter doesn't know what a name refers to, it raises a **name error**. The exact message may vary, depending on the context. For example, the error may occur if you misspell a name:

```
[8]: import maths # should be math
-----
↔-----  
ModuleNotFoundError Traceback (most recent)
    ↵call last)
Cell In[8], line 1
----> 1 import maths # should be math

ModuleNotFoundError: No module named 'maths'
```

or if you forget to indicate in which module the name is defined:

```
[9]: pi
-----
↔-----  
NameError Traceback (most recent)
    ↵call last)
Cell In[9], line 1
----> 1 pi

NameError: name 'pi' is not defined
```

By the way, if you want to try out your own examples, you should create a checkpoint, copy my code cell, paste a duplicate below the original cell, and then work on the pasted cell. If you don't want to keep your examples, revert to the checkpoint, or simply delete the pasted cell.

2.2 Arithmetic operations

This section lists the most common arithmetic operations used in M269. Further operations may be introduced in later chapters, as needed. I first introduce the ADT operations and then the corresponding Python operations.

The mathematical or Python symbols that represent operations are called **operators**, and the values the operator applies to are the **operands**. For example, in $3 + 4$, value 3 is the left operand and value 4 is the right operand of the operator $+$.

2.2.1 On real numbers

The most common operations of the real number ADT are **addition**, **subtraction**, **multiplication** and **division**. The corresponding operators are $+$, $-$, \times and $/$. The corresponding results are called the **sum**, **difference**, **product** and **quotient** of the two operands.



Info: Other texts may write multiplication as $x \cdot y$, or even just xy , and division as $x \div y$.

Division isn't defined when the right operand is zero. For example, $3 / 0$ should be the quotient z such that $z \times 0 = 3$ because multiplication is the inverse of division. But there's no such z : any number multiplied by zero gives zero, not 3.

The **negation** operation, written $-x$, converts a negative number into a positive one and vice versa. We have $-0 = 0$ and $-(-x) = x$ for any real number x .

The **maximum** and **minimum** operations compute the largest and smallest of a series of numbers, respectively. Because the operation can take a variable number of operands, it's written in functional notation, with the name of the operation (max or min, in this case) preceding the operands, which are separated by commas and enclosed within parentheses (round brackets). For example, $\max(3, \pi, -0.5) = \pi$, and $\min(3, \pi, -0.5) = -0.5$.

The **floor** operation, written $\text{floor}(x)$, rounds x down. For example, $\text{floor}(0.7) = 0$ and $\text{floor}(-0.2) = -1$.

2.2.2 On integers

All the above operations are also part of the integer ADT, because every integer is a real number too, although it's a bit pointless to round an integer.

Addition, negation and the other operations produce an integer when applied to integers, except for division: dividing two integers may result in a real number, e.g. $2 / 3$ isn't an integer but $6 / 3$ is.

We will use the **exponentiation** or **power** operation, written x^y , only with integer operands, and when **exponent** y isn't negative. Under these conditions, the operation multiplies the **base** x with itself y times, e.g. $-2^3 = -2 \times -2 \times -2 = -8$. We define $x^0 = 1$ for all integers x .

The **modulo** operation, written $x \bmod y$, computes the remainder of dividing integer x by integer y . The modulo is undefined for $y = 0$. For example, dividing 5 by 2 results in 2 with a remainder of 1, so $5 \bmod 2 = 1$, whereas dividing 9 by 3 results in 3 without a remainder, so $9 \bmod 3 = 0$. If $x \bmod y = 0$ then we say that x is a **multiple** of y (or is **divisible** by y) and vice versa y is a **factor** or **divisor** of x .



Info: MU123 Unit 3 Section 1 introduces multiples, factors and powers.

Unfortunately, for negative operands, there are various definitions of modulo. M269 uses this one: $x \bmod y = x - \text{floor}(x / y) \times y$. The definition also works for positive operands. Here are some examples:

- $5 \bmod 2 = 5 - \text{floor}(5/2) \times 2 = 5 - 2 \times 2 = 1$
- $-5 \bmod 2 = -5 - \text{floor}(-5/2) \times 2 = -5 - (-3 \times 2) = 1$
- $5 \bmod -2 = 5 - \text{floor}(5/-2) \times -2 = 5 - (-3 \times -2) = -1$

Exercise 2.2.1

An integer is **even** if it's a multiple of two, otherwise it's **odd**. For example, 0, 2 and -2 are even, but 1 and -1 are odd. Which operation would you use, and how, to check if a given integer x is even or odd?

Answer

2.2.3 On int and float

The following operations are defined for both integers and floats.

Python uses `+`, `-` and `/` for addition, subtraction, negation and division, but the multiplication operator is `*`. The result of division is always a float.

```
[1]: 6 / 3  
[1]: 2.0
```

For the other operations, if both operands are integers, so is the result; otherwise, it's a float.

```
[2]: 2 + 3  
[2]: 5
```

```
[3]: 2.0 + 3  
[3]: 5.0
```

```
[4]: 2 * -3  
[4]: -6
```

The minimum and maximum operations are written as in mathematics. Here are the earlier examples, now in Python.

```
[5]: import math  
      max(3, math.pi, -0.5)  
[5]: 3.141592653589793
```

```
[6]: min(3, math.pi, -0.5)  
[6]: -0.5
```

We have to import a module in each notebook that uses it, but once a module is imported, it's available for all subsequent code cells in the same notebook.

2.2.4 On float

We'll apply the floor operation to floats only, although it can be applied to integers too. The Python module `math` implements the operation.

```
[7]: math.floor(1.5)
```

```
[7]: 1
```

```
[8]: math.floor(-1.5)
```

```
[8]: -2
```

2.2.5 On int

Python's **floor division** is equivalent to division followed by rounding down. The operation is only defined on integers. The operator is two slashes.

```
[9]: 2 / -3      # one slash: float division
```

```
[9]: -0.6666666666666666
```

```
[10]: 2 // -3     # two slashes: floor division
```

```
[10]: -1
```

```
[11]: 2 // 3
```

```
[11]: 0
```

Can you explain the last two results?

(Remember that a long thin line is a ‘stop reading and think’ sign.)

We have $2 / -3 = -0.66\dots$, which rounds down to -1 , whereas $2 / 3 = 0.66\dots$ rounds down to 0 .



Info: If you're studying M250, note that Python's integer division is different from Java's. Python uses floor division whereas Java uses truncated division, i.e. removes the decimal part, so 2 divided by -3 is -1 in Python but 0 in Java.

As mentioned before, we'll use the modulo and power operations on integers only. In Python, the corresponding operators are `%` and `**`.

```
[12]: 5 % 2
```

```
[12]: 1
```

```
[13]: -5 % 2
```

```
[13]: 1
```

```
[14]: 5 % -2
```

```
[14]: -1
```

```
[15]: -2 ** 3
```

```
[15]: -8
```



Info: Python's and Java's modulo operations also differ, e.g. $-5 \text{ modulo } 2$ is 1 in Python and -1 in Java. Python uses the definition shown earlier, based on floor division, whereas Java defines the modulo in terms of truncated division.

2.2.6 Mistakes

Chapter 1 showed that some code cells may depend on the execution of earlier code cells. In this notebook, you'll get a name error if you run any of the cells with `floor` without having first executed the earlier cell with `import math`.

Division, floor division and the modulo are undefined if the right operand is zero. The interpreter raises an error in such cases.

```
[16]: 2 % 0
```

```
-----
↔-----  

ZeroDivisionError                                Traceback (most recent)
  ↗call last)
Cell In[16], line 1
----> 1 2 % 0

ZeroDivisionError: integer modulo by zero
```

If you want, you can duplicate the above cell and replace the modulo operation by division or floor division.

If you leave a space between the slashes of the floor division operator, or the asterisks of the power operator, then the interpreter thinks you forgot an operand between two consecutive divisions or multiplications, and reports a generic **syntax error**.

```
[17]: 2 / / 3
```

```
Cell In[17], line 1
  2 / / 3
  ^
SyntaxError: invalid syntax
```

The little caret (^) points to where the interpreter detected the error. The source of the error is somewhere before that point, sometimes even in an earlier line of code.

A common mistake is to think that the minus sign is part of the literal. In fact, it's the negation operator, and this has a subtle side effect.

```
[18]: -2 ** 0      # expecting x^0 = 1 for all x  
[18]: -1
```

Why isn't the result 1? Do the laws of maths not apply to Python? Read on for the solution to this mild cliffhanger...

2.3 Expressions

An **expression** is a single value, like 3 or 2.4, a single variable, like x , or an operator applied to subexpressions. We can only apply an operator if we know the values of its operands. So, to compute the value of an expression we must first evaluate its subexpressions to obtain their values. We also need to know the value of each variable that occurs in the expression. The following examples assume the variables have already been replaced with their values.

Subexpressions that are not single values or variables are enclosed in parentheses. For example, the expression $(3 + 4) \times 5$ consists of a multiplication applied to subexpression $3 + 4$ (the left operand) and subexpression 5 (the right operand), which is a single value. In turn, subexpression $3 + 4$ is the addition of two values. The value of the expression is $(3 + 4) \times 5 = 7 \times 5 = 35$. By contrast, the value of $3 + (4 \times 5)$ is 23. Both expressions have the same operators, but the subexpressions differ. Hence the order of the operations also differs and so do the results.

Writing parentheses around each subexpression gets tedious for long expressions, so mathematicians invented conventions to reduce the need for parentheses. The conventions are based on two concepts: **precedence** and **associativity**.

The precedence of an operation indicates its order of evaluation in relation to other operations. For example, if multiplication has higher precedence than addition, then $3 + 4 \times 5 = 3 + (4 \times 5)$, but if addition has the higher precedence of the two, then $3 + 4 \times 5 = (3 + 4) \times 5$.

Operations with the same precedence are **left-associative**, i.e. executed from left to right, or **right-associative**, i.e. executed right to left. For example, if multiplication and addition have the same precedence and are left-associative, then $3 + 4 \times 5 = (3 + 4) \times 5$, but if they're right-associative, then $3 + 4 \times 5 = 3 + (4 \times 5)$.

The precedence and associativity of operations changed throughout the history of mathematics, and it varies among authors, programming languages, spreadsheet apps and calculators. It's a jungle out there... M269 follows Python's precedence and associativity rules, so that mathematical expressions can be translated directly to Python. The next table lists the operations from highest to lowest precedence, and indicates the associativity for operations at the same precedence level.

Operation	Associativity
brackets	left
exponentiation	right
negation	right
multiplication, division, floor division, modulo	left
addition, subtraction	left

If we consider brackets as operators, then bracketed expressions have the highest precedence and are evaluated ‘inside out’ (most nested first), because subexpressions must be evaluated before the operator is applied. Bracketed expressions at the same nesting level are evaluated left to right. The floor operator and any operation written in functional notation are also bracketed expressions.

With these rules, an expression represents a calculation algorithm: an expression consists of unambiguous instructions (the arithmetic operations) structured according to the precedence and associativity rules, which make it possible to evaluate an expression as a step-by-step procedure.

If the expression includes an operation on invalid operands, then the algorithm stops and the expression is considered **undefined**: it can’t be evaluated. If the expression is written in Python, then the interpreter, which is carrying out the evaluation algorithm, stops with some error message (usually division by zero).

The algorithm can be ‘shown’ by making the precedence and associativity rules explicit with parentheses and then rewriting the expression step by step until a value is obtained or until a value can’t be obtained. Some examples:

- $2^{3^2} = 2^{(3^2)} = 2^9 = 512$
- $3 / 0^5 = 3 / (0^5) = 3 / 0$: this expression is undefined
- $2 + - - 5 = 2 + (-(-(-5))) = 2 + (-5) = -3$
- $3 + 4 \times 5 = 3 + (4 \times 5) = 23$
- $3 - 4 - 5 = (3 - 4) - 5 = -6$
- $7 - \text{floor}(3/4 + 0.5) \bmod 2 = 7 - (\text{floor}((3/4) + 0.5) \bmod 2) = 7 - (\text{floor}(0.75 + 0.5) \bmod 2) = 7 - (\text{floor}(1.25) \bmod 2) = 7 - (1 \bmod 2) = 7 - 1 = 6$
- $-2^0 = -(2^0) = -1$

The final example explains the mystery at the end of the previous section: exponentiation has the highest precedence, so the negation is evaluated last. The minus sign is not part of the number, it’s an operator, and therefore subject to precedence and associativity rules. We have to put brackets to force our intended order of evaluation.

[1]: `(-2) ** 0`

[1]: 1

Negation has higher precedence than the other arithmetic operations, so $-2 + 3$ evaluates to 1 and not $-(2 + 3) = -5$.

Whenever you have an expression with exponentiation and negative numbers, I recommend adding parentheses to show explicitly how it should be evaluated, even if they are not necessary, as for example in `- (3 ** 4)`.

2.3.1 Mistakes

Since most arithmetic operators are left-associative, it's easy to forget that exponentiation is right-associative, which may lead to different results from what we're expecting.

```
[2]: 2**3**2 # this is evaluated right to left
```

```
[2]: 512
```

```
[3]: (2**3) ** 2 # use parentheses to force left-to-right evaluation
```

```
[3]: 64
```

To show clearly your intentions and make code easy to understand, I think it's best to write the first example as `2 ** (3 ** 2)`, even though the parentheses are redundant, because it reminds the reader how the expression is evaluated.



Note: When the expression involves exponentiation, use parentheses, even if they're not necessary.

Exercise 2.3.1

In the previous section I wrote $-2^3 = -2 \times -2 \times -2 = -8$. Is this correct, considering the precedence and associativity rules now introduced? If yes, explain why; if not, show how -2^3 is evaluated.

Hint Answer

2.4 Assignments

As mentioned in the previous section, if an expression does involve variables, then we must know their values in order to evaluate the expression. A variable is said to be undefined if no value has been assigned to it. Expressions involving undefined variables are undefined too.

To define a variable, we use the **assignment** instruction, which evaluates an expression and assigns its value to a variable. In M269, when writing algorithms in English, assignments are of the form

let *variable* be expression

with variable names in italics, so that they stand out. In Python, assignments are of the form `variable = expression`.

Python allows us to abbreviate `x = x + y` as `x += y`, `x = x * y` as `x *= y`, and so on. The most frequently used forms are `x += 1`, `x -= 1`, `x *= 2` and `x // 2` to increment, decrement, double and halve an integer variable.



Note: I won't use abbreviated assignments in the book, but you can use them in the TMAs.

2.4.1 Algorithms

In this chapter, an algorithm is a sequence of assignments. Here's a simple algorithm that computes the age of a person in a particular year as the difference between that year and their birth year. We write algorithms as numbered lists to show the order of steps.

1. let `year` be 2020
2. let `birth` be 2010
3. let `age` be `year - birth`

The translation to Python is:

```
[1]: year = 2020  
      birth = 2010  
      age = year - birth
```

The previous cell doesn't display a value, because assignments aren't like arithmetic operations: they don't produce an output. In Jupyter notebooks, the value of an expression (which may be a single variable) is displayed only if the expression is the last or only line in a code cell.

```
[2]: year = 2020  
      year           # not last line, so it isn't displayed  
      birth = 2010  
      age = year - birth  
      age           # last line, so it's displayed
```

```
[2]: 10
```

To display the value of an expression at any point in an algorithm, we'll use the instruction

print expression

in English, and `print(expression)` in Python.

```
[3]: year = 2020  
      print(year)  
      birth = 2010  
      age = year - birth  
      age  
  
      2020
```

[3] : 10

2.4.2 Names

Names can consist of multiple words. In English and other languages, we separate multiple words with spaces. In Python, they are joined together with underscores.

1. let *current year* be 2020
2. let *year of birth* be 2010
3. let *age* be *current year* – *year of birth*
4. print *age*

[4] :

```
current_year = 2020
year_of_birth = 2010
age = current_year - year_of_birth
print(age)
```

10

Names can include digits, which may come in handy if you need a few variables with unrelated values, e.g. *year1* and *year2*, or *year 1* and *year 2*, and correspondingly *year1*, *year_1*, etc.

A period can only be used to separate names of different things, like in `math.pi` to separate the names of the module and of the variable.

You may see Python code that uses mixed-case variable names, like `currentYear`, `yearOfBirth`, `Year_of_Birth`, etc. Although they're all valid names according to Python's syntax, in M269 we follow the convention of writing variable names in lowercase, with underscores to separate words.



Note: Following conventions makes your code easier to understand by others.



Info: Writing names in lowercase with underscores is known as snake case. The Python coding style conventions are given in Python Enhancement Proposal (PEP) 8.

Descriptive names make algorithms much easier to understand. Use full English words. Avoid abbreviations, like `yob = 2010`, and single-letter names, like `y = 2020`.



Note: Use descriptive names that reflect what the variables represent.

It's time to apply what you learned in this and the previous section. The next exercises ask you to write assignments and expressions that have the brackets, if necessary, in the right places. As mentioned in Chapter 1, I assume you know how to work with percentages.

Exercise 2.4.1

Many countries impose a sales tax or value-added tax (VAT) on various products and services. Write an algorithm in English that:

1. sets the price of a product to an integer or real value of your choice
2. sets the tax rate to 20%
3. computes the total price, including the tax to be paid
4. prints the total price

Answer

Exercise 2.4.2

Translate the previous algorithm to Python code. Don't forget to press Tab to autocomplete names. If you get errors when executing your code, check the next subsection.

```
[5]: # replace this by your code
```

Answer

2.4.3 Mistakes

An assignment's expression can only refer to previously assigned variables, otherwise the expression is undefined. The following sequence doesn't define variable *age* because *birth* is undefined.

1. let *year* be 2020
2. let *age* be *year* – *birth*

If the above algorithm were written and executed in Python, the interpreter would stop in the second step with a name error for *birth*.

If your algorithm for an exercise is calculating a wrong final value, consider using print statements to display the intermediate values. This helps you narrow down the source of the error.

If you forget the underscores or periods between the words of a name in Python, you get a syntax error.

```
[6]: current year = 2020      # missing underscore
      Cell In[6], line 1
      current year = 2020      # missing underscore
      ^
```

(continues on next page)

(continued from previous page)

```
SyntaxError: invalid syntax
```

Every character matters, so *current year*, *currentyear* and *currentYear* are different names. In Python, if you misspell a name, you get a name error.

```
[7]: currentyear      # current_year is defined, currentyear isn't
-----
↑-----  
NameError                               Traceback (most recent...)
    ↗call last)
Cell In[7], line 1
----> 1 currentyear      # current_year is defined, currentyear isn't

NameError: name 'currentyear' is not defined
```



Note: If you get a name error then you either misspelt the name, didn't assign a value to it (e.g. you forgot to run a particular cell), didn't indicate the module defining the name, or didn't import the module.

Hyphens, apostrophes and punctuation other than periods aren't allowed in names. If you use them in Python, you'll get a (possibly strange) error.

```
[8]: Bob's_phone_number  # curved apostrophe
      Cell In[8], line 1
      Bob's_phone_number  # curved apostrophe
      ^
SyntaxError: invalid character '' (U+2019)
```

```
[9]: Bob's_phone_number  # straight apostrophe
      Cell In[9], line 1
      Bob's_phone_number  # straight apostrophe
      ^
SyntaxError: unterminated string literal (detected at line 1)
```

```
[10]: left-associative
-----
↑-----  
NameError                               Traceback (most recent...)
```

(continues on next page)

(continued from previous page)

```
↳call last)
Cell In[10], line 1
----> 1 left-associative

NameError: name 'left' is not defined
```

Why is the last example a name error rather than a syntax error?

It's a name error because the hyphen is interpreted as the difference operator between two numeric variables, but neither has been assigned a value.

2.5 Functions in mathematics

In M269, you won't be asked to develop complete applications (with user interface, database, etc.) but rather to design and implement algorithms for individual operations on some data, e.g. to add the VAT to a given price.

In M269, most operations are functions as defined in mathematics, so let's recap them.

A function has a set of allowed input values and a rule for calculating exactly one output value from the input values, i.e. a function never produces two different outputs for the same input values. A function doesn't modify its inputs.

For example, the function to calculate the length of the circumference of a circle with radius r can be defined as $g(r) = 2 \times \pi \times r$ ($r > 0$), where:

- g is the name of the function
- r is the input variable
- $r > 0$ indicates that only positive real numbers are allowed
- $2 \times \pi \times r$ is the rule for calculating the output value.



Info: MST124 Unit 3 Section 1.2 and MU123 Unit 6 Section 3.1 define functions. The example is from MST124 Unit 3 Section 1.3.

In MU123 and MST124, the input and output values are always real numbers, and therefore a function rule can be stated as a formula. M269 doesn't use just one data type and the functions to be implemented are more complicated than a single formula, so we need a more general notation to define them. We shall use this template:

Function: the name of the function

Inputs: the name and data type of each input

Preconditions: any conditions on the inputs

Output: the name and data type of the result

Postconditions: how the output relates to the inputs

If you wish, you can use the singular ('input', 'precondition' and 'postcondition') when there's only one input or condition, but I always use the plural, to reuse the same template for different functions.

Here's how function $g(r) = 2 \times \pi \times r$ ($r > 0$) could be defined with this template:

Function: g

Inputs: r , a real number

Preconditions: $r > 0$

Output: $g(r)$, a real number

Postconditions: $g(r) = 2 \times \pi \times r$

The data types *must* be stated: we can't assume they're always real numbers. Strictly speaking, the output's data type can be inferred from the formula in the postconditions, but it's clearer to state it explicitly. The types are usually ADTs, so that the function can be implemented with different programming languages. The pre- and postconditions state what must be true before and after the function is applied. The preconditions define the allowed input values. For simple numeric functions, the postcondition states that the output is equal to the value of an expression over the input variables.

A better definition for the circumference function uses descriptive names, so that the reader doesn't have to figure out what the function computes from the postconditions. Moreover, in M269 we use a variable for the computed output value. A clearer definition is, for example:

Function: circumference

Inputs: radius, a real number

Preconditions: radius > 0

Output: length, a real number

Postconditions: length $= 2 \times \pi \times \text{radius}$

At this point you may be wondering what's the point of this far more verbose notation than simply writing $g(r) = 2 \times \pi \times r$ ($r > 0$). The main advantage is that the template explicitly shows the different parts of a function and thus serves as a 'thinking scaffold' when defining your functions. Filling the template prompts you to think about:

- the name of the function, to reflect what it does
- how many inputs are needed and what they represent
- any invalid input values
- how the output is obtained from the inputs.

2.5.1 Example

To illustrate how the template can guide your thinking, let me go through it to define a function to compute the volume of a brick.

The function name should be descriptive but not too long, and capture *what* the function does or produces, not *how* it does it (that's the algorithm's job). One possible name is:

Function: brick volume

Next, the inputs. The statement 'compute the volume of a brick' doesn't specify them. I have to make a reasonable decision. What values do I need to compute the volume? I need the brick's width, length and height. What are their types? This will depend on the measurement units and precision. In real life I'd contact the client and ask them for the units and precision in which the input data is provided. In the absence of a client's wishes, I always prefer integers to real numbers due to the limitations of representing real numbers in Python. I will thus assume that the input values are integers. To achieve a reasonable measuring precision, I assume the unit is millimetres.

Inputs: *length*, an integer; *width*, an integer; *height*, an integer

You may separate inputs with semicolons or with bullet points, as you prefer.

Inputs:

- *length*, an integer
- *width*, an integer
- *height*, an integer

Next I must think if any input values are not allowed. Like it was the case with the circle's radius, a brick cannot have zero or negative length, width or height.

Preconditions: $length > 0$; $width > 0$; $height > 0$

I assumed that the dimensions are given in millimetres. I must state that, so I need to add:

Preconditions: *length*, *width* and *height* are in millimetres



Note: Always state your assumptions explicitly.

Next comes the output. Without a client telling me what the output's unit and precision are, I must make a sensible decision. I choose to have an integer value, in cubic millimetres, to not lose precision. Again, I state the data type and the units separately.

Output: *volume*, an integer

Postconditions: *volume* is in cubic millimetres

Finally, I add the most important postcondition, which relates the output to the inputs.

Postconditions: $volume = length \times width \times height$

Here's the whole definition:

Function: brick volume

Inputs: $length$, an integer; $width$, an integer; $height$, an integer

Preconditions:

- $length > 0$; $width > 0$; $height > 0$
- $length$, $width$ and $height$ are in millimetres

Output: $volume$, an integer

Postconditions:

- $volume = length \times width \times height$
- $volume$ is in cubic millimetres

Exercise 2.5.1

Fill out the template below to define a function that, given the price of a product or service and a VAT rate, calculates the total price, including VAT at the given rate. Some products and services are not subject to VAT. The VAT rate is always less than 100%.

When editing the next cell, keep the backslashes at the end of the lines to enforce the line breaks.

Function:

Inputs:

Preconditions:

Output:

Postconditions:

Hint Answer

2.5.2 Algorithms

To implement a function, we need a sequence of one or more assignments that take the input values and produce the output value. For simple numeric functions, the algorithm may be a single assignment based on the formula in the postconditions. For the circumference function, the algorithm can be simply:

1. let $length$ be $2 \times \pi \times radius$

After this assignment is executed, the value of $length$ is equal to the value of $2 \times \pi \times radius$, i.e. the postcondition is satisfied. An alternative algorithm is:

1. let $diameter$ be $2 \times radius$
2. let $length$ be $\pi \times diameter$

This one also satisfies the postcondition $length = 2 \times \pi \times radius$.

Function definitions don't just help you think about the problem at hand. They are crucial to check if an algorithm is **correct**: an algorithm correctly implements a function if for all inputs

that satisfy the preconditions it produces an output satisfying the postconditions. A function is considered undefined for invalid inputs, i.e. there's no meaningful output for invalid inputs, so for an algorithm to be correct it doesn't matter what it does when it gets inputs that break the preconditions.

2.5.3 Mistakes

The preconditions state what is true *before* the function is applied. At that point there's no output yet, so the output variable must never appear in the preconditions.

Don't feel forced to write preconditions. Some functions don't have any, e.g. the floor function can be applied to any real number. In such cases, for the moment leave that part of the template empty. The next chapter introduces a better way to indicate there are no preconditions.

Functions should be as generally applicable as possible, so don't constrain the input values unnecessarily. For example, the precondition $length \geq width$ isn't needed to correctly compute a brick's volume and so should be left out.

By the way, the symbols \leq , \geq and \neq occur often in preconditions, so try to find out if they can be easily typed on your keyboard. On a Mac, they're obtained by pressing Alt and $<$, $>$ or $=$, respectively.

While a function may not have preconditions, it always has one or more postconditions. They effectively define the function. The postconditions must involve *all* input and output variables. If an input variable doesn't appear in the postconditions, then you're stating it's not needed to compute the output: so why should it be an input?

When writing an algorithm, especially a longer one, it's easy to forget a variable in an expression or to forget an assignment. Check that each expression only refers to previously assigned variables or to the input variables. This means that the first assignment in the sequence can only use the input variables, as no other variable is defined at that point.

Each variable should be assigned only once and the output variable should be assigned last. If this isn't the case, then your sequence of assignments has room for improvement. You may be doing unnecessary work or using meaningless variables, like this:

1. let *diameter* be 2
2. let *diameter* be $diameter \times radius$
3. let *length* be $\pi \times diameter$
4. let *pi* be π

This is a correct algorithm (it satisfies the postcondition), but a rather poor one. Step 4 is unnecessary and the variable name in step 1 is meaningless because the diameter isn't two.

The next algorithm is also correct, but there's no meaning to twice the value of π , whereas twice the radius is a meaningful concept (the circle's diameter).

1. let *double pie* be $2 \times \pi$
2. let *length* be *double pie* $\times radius$

If you can't give the variable on the left-hand side of an assignment a meaningful name, then the right-hand expression may not make much sense either. It's probably time to rewrite the assignments.



Note: Write assignments that compute meaningful values and name the variables accordingly.

2.6 Functions in Python

A function definition of the form:

Function: name

Inputs: *input*, input type

Preconditions: condition 1; condition 2

Output: *output*, output type

Postconditions:

- *output* = expression
- another condition

is translated to a Python function definition of the form:

```
def name(input: input_type) -> output_type:  
    """Prescribe what the function computes.  
  
    Preconditions: condition 1; condition 2  
    Postconditions:  
        - the output is expression  
        - another condition  
    """  
    algorithm  
    return output
```

For example,

Function: circumference

Inputs: *radius*, a real number

Preconditions: *radius* > 0

Output: *length*, a real number

Postconditions: *length* = $2 \times \pi \times \text{radius}$

is implemented in Python like so:

```
[1]: import math

def circumference(radius: float) -> float:
    """Return the length of the circumference for the given radius.

    Preconditions: radius > 0
    Postconditions: the output is 2 * π * radius
    """

    length = 2 * math.pi * radius
    return length
```

Defining a function doesn't produce any output (only applying a function does), but the number on the left shows that the cell was executed, i.e. the function has been successfully defined.

A Python function has three parts: the header, the **docstring** (short for documentation string) and the body. (Technically, the docstring is part of the body, but I prefer to separate them.) The docstring and body are indented with respect to the header to indicate that they belong to the function. The conventional indentation is four spaces, as in the above example. Let's look more closely at the header and body. (I'll describe docstrings in the next subsection.)

The header starts with the `def` keyword and ends with a colon. The header defines the function's name and its **parameters**, i.e. input variables. The convention is to write function names in lower case, like variable names. Parameters are enclosed in parentheses and separated with commas. The header also includes **type annotations** to indicate the type of each parameter and the type of the output, as shown above.

The body contains the algorithm that computes the output value. Contrary to our template, the function header doesn't indicate the output's name, so after the algorithm (a sequence of assignments) we need a **return statement** to indicate which of the assigned variables corresponds to the output. A return statement ends the execution of the body: the interpreter returns the value of the variable following the `return` keyword to the code that called the function.

To **call** a function, write its name followed by the **arguments**, i.e. the actual input values, within parentheses. A function call is an expression because a function defines an operation on some given data. The following cell has a single expression, so the returned value is displayed.

```
[2]: circumference(1) # calculate for radius = 1
[2]: 6.283185307179586
```



Info: TM112 Block 2 Part 4 introduces functions in Python. In TM112 and other texts, the input variables (parameters) are also called formal arguments while the input values are called the actual arguments.

Actually, the `return` keyword may be followed by any expression, not just a variable name. The interpreter evaluates the expression and returns its value. Here's a shorter alternative

version. (Remember that we only import a module once per notebook.)

```
[3]: def circumference(radius: float) -> float:  
    """Return the length of the circumference for the given radius.  
  
    Preconditions: radius > 0  
    Postconditions: the output is 2 * π * radius  
    """  
  
    return 2 * math.pi * radius
```

This function has the same name as an earlier function, so this definition overrides the previous one. From now on, any calls to `circumference` execute the second version, without the `length` variable. Since both function definitions are equivalent, it doesn't really matter. When writing different versions of a function for exercises, remember that only the last executed function definition is active.

2.6.1 Documentation

Docstrings are enclosed in three double quotes ("'"). The first line is a brief statement prescribing what the function does, usually starting with the word 'return'. After a blank line, more details may follow. If the docstring occupies several lines, then the closing quotes are on their own line.



Info: Python's docstring conventions are described in [PEP 257](#).

In M269, we write pre- and postconditions in the docstring, so that users know what input values the function expects and what value it returns. In Python and other languages, function headers have names for the inputs but not for the output. I suggest using simply 'the output' in the docstring. The important thing is for the header and the docstring to contain, together, the same information as the template.

Docstrings are optional in Python, i.e. they're not needed for functions to work, but are mandatory in M269. The rationale is to get you into the good professional habit of documenting your code, so that those who use it or have to modify it know what it does.



Note: Always document your code.

Python's `help` function displays the header and docstring of a function, given its name. This works with your and Python's functions. It may come in handy if you forget what a function does.

```
[4]: help(circumference)
```

```
Help on function circumference in module __main__:
```

```
circumference(radius: float) -> float
    Return the length of the circumference for the given radius.

    Preconditions: radius > 0
    Postconditions: the output is  $2 * \pi * \text{radius}$ 
```

```
[5]: help(max)
```

```
Help on built-in function max in module builtins:
```

```
max(*)
    max(iterable, *, default=obj, key=func) -> value
    max(arg1, arg2, *args, *, key=func) -> value
```

With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty.

With two or more arguments, return the largest argument.

As the docstring for `max` shows, built-in Python functions may have optional parameters that we won't use in M269.

2.6.2 Mistakes

Python functions are the most complex construct seen so far, and thus provide ample opportunity for mistakes: forgetting the colon at the end of the header, forgetting a comma between consecutive parameters, misspelling a parameter name in the function body, etc.

Using a keyword as an identifier is also a syntax error:

```
[6]: return = 2 * math.pi * radius
```

```
Cell In[6], line 1
    return = 2 * math.pi * radius
        ^
SyntaxError: invalid syntax
```

Referring to a parameter outside the function body is a name error:

```
[7]: diameter = 2 * radius
```

```
-----  
←----  
NameError
```

Traceback (most recent)

(continues on next page)

(continued from previous page)

```
→call last)
Cell In[7], line 1
----> 1 diameter = 2 * radius

NameError: name 'radius' is not defined
```

A double quote is *one* character, not two single quotes: note the difference between " and ''.

```
[8]: """This is a docstring."""           # note the syntax colouring
[8]: 'This is a docstring.'

[9]: '''''This is not a docstring.''''''    # different syntax colouring

    Cell In[9], line 1
    '''''This is not a docstring.''''''    # different syntax
→colouring
^
SyntaxError: invalid syntax
```

If you forget to indent the docstring or the body, you get an error.

```
[10]: def circumference(radius: float) -> float:
        """Return bla bla bla ...
        """
        return 2 * math.pi * diameter

    Cell In[10], line 2
    """Return bla bla bla ...
    ^
IndentationError: expected an indented block after function
→definition on line 1
```

```
[11]: def circumference(radius: float) -> float:
        """Return bla bla bla ...
        """
        return 2 * math.pi * diameter

    Cell In[11], line 4
        return 2 * math.pi * diameter
    ^
SyntaxError: 'return' outside function
```

The second error has subtle consequences. If you call the function, you get no value back because the return statement wasn't considered part of the function.

```
[12]: circumference(1)
```

This means that the interpreter still defined the function, even though it had no return statement! For example, the following definition doesn't raise any error.

```
[13]: def circumference(radius: float) -> float:
    """Return ...
    diameter = 2 * radius
    length = math.pi * diameter
```



Note: If a function doesn't return a value, then the return statement is missing or wrongly indented.

Exercise 2.6.1

Here's the brick volume function definition again:

Function: brick volume

Inputs: *length*, an integer; *width*, an integer; *height*, an integer

Preconditions:

- *length* > 0; *width* > 0; *height* > 0
- *length*, *width* and *height* are in millimetres

Output: *volume*, an integer

Postconditions:

- *volume* = *length* × *width* × *height*
- *volume* is in cubic millimetres

Translate it to Python. I've given you a head start.

```
[14]: def brick_volume(length: int):
    """Return ...
    """
```

Run the previous cell to define the function, then uncomment and run the next cell.

```
[15]: # brick_volume(2, 3, 4)    # the output should be 2 * 3 * 4 = 24
```

Hint Answer

Exercise 2.6.2

Here's a definition of the total price function.

Function: total price

Inputs: *price*, a real number; *vat rate*, a real number

Preconditions: $price > 0$; *price* is in euros; $0 \leq vat\ rate < 1$

Output: *total*, a real number

Postconditions: $total = price \times (1 + vat\ rate)$; *total* is in euros

Write the corresponding Python function in the next code cell. End the code cell with a function call to check the definition has no errors.

```
[16]: # replace this by a function definition  
  
# replace this by a function call
```

Answer

2.7 Complexity

We want algorithms to be correct *and* fast, especially on large inputs. The run-time of an algorithm, implemented as a Python function, depends on the hardware, operating system and Python interpreter we're using, and whether other processes are running in the background, like checking for software updates.

Computer scientists found a way of talking about algorithms that is independent of all these factors. Instead of getting bogged down with the exact run-times for particular input values, we look at how the run-times increase for ever-larger inputs. In other words, what we really want to know is how well (or not) an algorithm copes with growing inputs.

2.7.1 Constant complexity

The algorithms that best cope with growing inputs are those where the run-time stays roughly the same, no matter how small or large the input is. Such algorithms are said to have constant run-time or **constant complexity**. The term 'constant' doesn't mean that the run-time stays *exactly* the same for all inputs: it means that it doesn't grow.

The **complexity** of an algorithm is the growth rate of its run-times as inputs get larger, when executed on the same computational environment (hardware, operating system, programming language and interpreter). The complexity is *not* about how fast the algorithm runs. For example, an addition algorithm that would take a whole day to find out the sum of 3 and 4 but also takes one day (in the same environment) to add two 500-digit numbers would have constant complexity. A constant complexity algorithm may be slow, but it won't get slower for larger inputs.

A simple way to see if an algorithm has constant complexity is to implement the algorithm in some computational environment, run it with ever larger inputs, measure the run-times and see if they remain more or less the same. A better approach is to determine the complexity of an

algorithm before implementing it, from its English description. This prevents wasting effort in coding and testing algorithms that turn out to be inefficient. To determine the complexity of an algorithm we have to agree on the complexity of each operation it uses.

M269 covers general-purpose algorithms, not specialised ones that require humongous numbers with hundreds of digits, like in cryptography. Even though Python supports arbitrarily large integers, 64-bit integers and floats are large enough for our purposes.

```
[1]: (2**63) - 1 # largest 64-bit integer; 1 bit is for the sign
[1]: 9223372036854775807
```

Modern processors can do arithmetic operations on two 64-bit numbers with a single hardware instruction, so for M269's purposes we can assume that all *arithmetic operations* (except exponentiation, which I explain later) have constant complexity. We're *not* assuming that, for example, multiplication takes the same time as addition, but rather that adding 3 and 7 takes about the same time as adding 3 million and 7 million, and that multiplying 3 and 7 takes about the same time as multiplying 3 million and 7 million.

We also assume that assignments and return statements have constant complexity because the work required is always the same, no matter how small or large the value being named or returned is. To be clear, we're not assuming that `x = expression` or `return expression` always takes the same time, as that will depend on the expression. However, once the expression is evaluated, assigning the value to a name or returning the value is a constant-time operation.

If each instruction always takes some fixed amount of time, and the number of instructions is fixed, i.e. doesn't depend on the inputs, then the overall time the algorithm takes is also fixed. For example, $\text{floor}(x \times y / z)$ consists of three constant-time arithmetic operations, so the evaluation of the expression also takes constant time. Multiplication, division and computing the floor all take different times, but each takes a fixed time, independent of the values of its operands, so the overall time is also fixed.



Note: An algorithm that executes a fixed number of operations, each with constant complexity, has constant complexity.

The **Big-Theta notation** states the complexity in a concise and precise way. If the run-time is constant, we say that the algorithm has complexity $\Theta(1)$, or takes $\Theta(1)$ time, or has run-time $\Theta(1)$. The $\Theta(1)$ notation informally means ‘proportional to 1’, which is a roundabout way of saying ‘constant’ because a value that is proportional to a constant (1 in this case) is also constant. While constant complexity could also be written as $\Theta(2)$, $\Theta(57)$ or with any other fixed value, the convention is to write $\Theta(1)$.

2.7.2 Linear complexity

In primary school we learned an algorithm that adds two arbitrarily large integers digit by digit, from right to left, carrying over 1 from one addition to the next when necessary. Since adding two digits (possibly with a carry over) takes constant time, the time to add two integers is directly proportional to the number of digits of the longest integer, e.g. $222 + 88$ requires three

digit additions, which are (from right to left) $2 + 8$, $2 + 8 + 1$ carry over, and $2 + 1$ carry over. If the number of digits of the longest integer doubles, then addition will take double the time.

Algorithms where the run-time grows proportionally to the value or size of the inputs have **linear complexity** or take linear time. The **size** of an input is, strictly speaking, how much memory it occupies. Since the memory allocated to an integer may vary across computational environments, we use a proxy measure. For the purposes of M269, the size of integer n , written $|n|$, is the number of its decimal digits, e.g. $|102| = 3$.

If the run-time is constant, then it doesn't depend on the inputs, but for linear-time algorithms we have to state how their run-time exactly depends on the inputs. For example, the school algorithm for $x + y$ is linear in $\max(|x|, |y|)$, i.e. its run-time is proportional to the largest size of the two integers being added.

The Big-Theta notation $\Theta(\dots)$ indicates that an algorithm's run-time is proportional to ..., so we can simply state: the complexity of the school addition algorithm for integer inputs x and y is $\Theta(\max(|x|, |y|))$.

The school addition algorithm works for arbitrary large integers, but in M269 we only use 64-bit integers, which have at most 19 decimal digits (see the largest 64-bit above). So, even if computer hardware were to use the school algorithm for adding x and y , the complexity would be at most $\Theta(\max(|x|, |y|)) = \Theta(\max(19, 19)) = \Theta(19) = \Theta(1)$. In other words, while adding arbitrary large integers takes linear time, adding integers with a bounded number of digits (like 64 binary digits or 19 decimal digits) takes constant time, because the run-time won't grow beyond what it takes to process two operands that are the largest 64-bit numbers.

Likewise, we can expect subtraction, multiplication, division and modulo to take longer the more digits they need to process. However, by assuming that we will only deal with 64-bit numbers we can treat them all as constant-time operations.

Let's now consider exponentiation. (Remember that we *don't use negative exponents* in M269.) For integers x and y , with $y \geq 0$, we have $x^y = 1 \times x \times x \times \dots \times x$. Hence $x^0 = 1$ requires zero multiplications, $x^1 = 1 \times x = x$ requires one multiplication and in general x^y requires y multiplications. If each multiplication takes constant time and if y doubles in value (not size!), then the number of multiplications (and therefore the run-time) also doubles. The exponentiation algorithm is therefore linear in the value of the exponent, not in the number of its digits. We write that the complexity of x^y is $\Theta(y)$.

If the complexity of exponentiation depended on the *size* of the exponent, then we'd know there would be at most 19 multiplications and we could treat exponentiation as a constant-time operation, like we did for the other arithmetic operations. But since the number of multiplications depends on the *value* of the exponent, even if its size remains fixed, e.g. at 4 decimal digits, the number of multiplications (and therefore the run-time) keeps growing, e.g. from 1000 to 9999.

Actually, it takes constant time to compute x^y when $y = 0$, because no multiplication is done. So the complexity of the algorithm varies: it's constant for $y = 0$ and linear for $y > 0$. When the complexity is different for small inputs, we just ignore it, because we're only interested in how an algorithm behaves for large inputs. So, we keep stating that the complexity of exponentiation is linear in the exponent's value, even though it's constant for one small exponent.

To sum up, $\Theta(e)$, where e is an expression involving zero or more of the input variables, means

that the run-time is proportional to e for large inputs. (It may or not be proportional to e for small inputs.)

Note that we assumed that multiplication takes constant time. In reality, as we keep multiplying with a 64-bit number x , at some point the intermediate result may not fit into 64 bits and we can't assume that each further multiplication takes constant time. So, for arbitrary integers x and y , exponentiation by repeated multiplication actually takes more than linear complexity. However, complexity analysis is a back of the envelope calculation to approximately predict the growth of run-times, so we're entitled to make some simplifying assumptions, as long as we clearly state them.

2.7.3 Mistakes

I wrote that e involves zero or more input variables because an algorithm's complexity either doesn't depend at all on the inputs (constant complexity) or depends on one or more of the inputs. The variables that appear in the complexity expression must always be some or all of the input variables, otherwise the complexity isn't defined. For example, if a function definition starts like this:

Function: secret operation

Inputs: $left$, an integer; $right$: an integer

then I can't write that an algorithm for this function has complexity $\Theta(x)$ or $\Theta(\max(l, r))$ or $\Theta(|y|)$ because none of those variables are defined: they don't refer to any of the inputs. I must write $\Theta(left)$ or $\Theta(\max(left, right))$ or $\Theta(|right|)$ or whatever the complexity is.



Note: Many texts always use the variable n in Big-Theta expressions, without making clear to what the variable refers. Don't follow their example.

Another common mistake is to confuse the size and the value of an integer. For example, if the complexity of x^y were $\Theta(|y|)$, then it would mean that the complexity is linear in the size of the exponent. If that were so, x^{44} would take double the time to compute as x^4 , because $|44| = 2$ and $|4| = 1$, when in fact it takes 11 times longer, because x^{44} requires 44 multiplications whereas x^4 requires four. The complexity of exponentiation is linear in the value (not the size!) of the exponent, i.e. it is $\Theta(y)$, not $\Theta(|y|)$.

Exercise 2.7.1

Here again is an algorithm for the circumference, where $radius$ is the input variable and $length$ is the output variable.

1. let $diameter$ be $2 \times radius$
2. let $length$ be $\pi \times diameter$

What is the complexity of this algorithm? State it in words and with Big-Theta notation.

Hint Answer

2.8 Run-times

We determine the complexity of an algorithm from its English description. Later, after implementing it in Python, we can measure the run-times for ever-growing inputs to check the actual growth rate against what the complexity analysis predicted.

The run-time depends on the hardware, operating system and Python interpreter that execute the code, so you'll get different timings from mine if you run the code cells in this notebook. The run-times also depend on what other processes the computer is executing, so they change every time we run the code.

We can measure the run-time of some code with the IPython command `%timeit`. Instructions starting with a percentage sign aren't part of the Python language: they're direct commands to the IPython interpreter.

The next cell will take some seconds to run (I'll explain why further below). While code is running, the notebook has an asterisk instead of a number in the brackets to the left of the code cell. Once the code finishes executing, the asterisk becomes a number.

```
[1]: %timeit 2 + 7
3.63 ns ± 0.0203 ns per loop (mean ± std. dev. of 7 runs, 100,000,
→000 loops each)
```

Adding 2 and 7 takes a few nanoseconds (abbreviated ns). The run-time varies each time the code cell is run, depending on the CPU load, so as I write this text I don't know in advance the run-time you will see. In the following I assume computing $2 + 7$ takes 8 nanoseconds.

Other abbreviations you may see in the run-time report are 's' (seconds), 'ms' (milliseconds) and ' μ s' (microseconds). One second is a thousand milliseconds; one millisecond is a thousand microseconds; one microsecond is a thousand nanoseconds. In other words, $1 \text{ ms} = 10^{-3} \text{ s}$, $1 \mu\text{s} = 10^{-6} \text{ s}$ and $1 \text{ ns} = 10^{-9} \text{ s}$.

Measuring a very short lapse of time is prone to significant measurement errors, so `%timeit` executes the code multiple times, measures the total run-time, and divides it by the number of iterations to get a more precise value. On my machine, the addition was computed 100 million times, but on yours it may have been fewer or more times: the IPython interpreter automatically chooses the number of loops. Finally, to reduce the effect of other processes running at the same time, `%timeit` runs everything seven times and takes the average.

Even though addition is very fast, running the code cell takes several seconds. Assuming addition takes 8 ns, the cell executes in $7 \text{ runs} \times 100,000,000 \text{ loops} \times 8 \text{ ns} = 7 \times 100 \times 8 \text{ ms} = 5,600 \text{ ms} = 5.6 \text{ s}$. Fortunately, we can reduce the time waiting for a result by setting the number of runs and loops with options `-r` and `-n`, respectively.

```
[2]: %timeit -r 5 -n 1000000 2 + 7
3.66 ns ± 0.0334 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
```

I reduce the number of runs to five and the number of loops to one million. Now the cell runs in milliseconds: $5 \times 1,000,000 \times 8 \text{ ns} = 5 \times 1 \times 8 \text{ ms} = 40 \text{ ms}$.

The time measured for addition may differ from previously, e.g. it may now be 7.5 ns or 8.5 ns instead of 8 ns, because reducing the number of runs and loops reduces accuracy. That's OK, as I'll explain in a minute.

Modern processors add two 64-bit numbers in hardware, so the very short time is not a surprise. Let's try some inputs that don't fit in 64 bits.

```
[3]: %timeit -r 5 -n 1000000 (2 ** 64 + 1) + (2 ** 64 + 2)
3.65 ns ± 0.0309 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
```

This took about the same time! That can't be right. Let's try a different way.

```
[4]: left = 2
right = 7
%timeit -r 5 -n 1000000 left + right
left = 2**64 + 1
right = 2**64 + 2
%timeit -r 5 -n 1000000 left + right
12.6 ns ± 0.357 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
24.3 ns ± 0.498 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
```

The times look right now: adding longer numbers takes more time, as we'd expect. Before, the interpreter figured out I was adding two constant values and pre-computed the sum, as it wouldn't change. I was just measuring the time to retrieve a value from memory, which is always the same, no matter how small or large the value is. The interpreter can't make such optimisations when adding variables because their values may change.



Note: When measuring the run-time of an expression, use variables instead of literals.

As I mentioned in the previous section, ‘constant time’ doesn't mean that *all* operations take the same time, but rather that *each* operation takes the same time for small and large inputs. For example, I expect that dividing two numbers takes longer than adding them.

```
[5]: left = 9876543210
right = 123456789
%timeit -r 5 -n 1000000 left + right
%timeit -r 5 -n 1000000 left // right
23.4 ns ± 0.232 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
29.1 ns ± 0.185 ns per loop (mean ± std. dev. of 5 runs, 1,000,000
→loops each)
```

We can also measure the run-time of functions we define.

```
[6]: def brick_volume(length: int, width: int, height: int) -> int:
    """Return the volume of a brick, given its dimensions.

    Preconditions: the dimensions are positive and in millimetres
    Postconditions: the output is in cubic millimetres
    """
    return length * width * height

l = 2
w = 3
h = 4
%timeit -r 5 brick_volume(l, w, h)
34.5 ns ± 0.145 ns per loop (mean ± std. dev. of 5 runs, 10,000,000 loops each)
```

I didn't set the number of loops, so the interpreter automatically sets it according to the run-time. The longer the code takes to run, the fewer loops are necessary to get a precise measurement. In this example, the interpreter on my machine 'only' made 10 million function calls in each run, whereas it had previously made 100 million additions per run.

2.8.1 Checking growth rates

To check the complexity of an operation, we measure the run-times for a series of inputs and look at the trend. That's why the actual run-time values don't matter, as long as we measure them consistently. What matters is how the run-times increase, as the inputs get larger. Often, we keep doubling the input size or value, as that's a good way to check if an operation has constant or linear complexity. Here's a little experiment for the addition operation.

```
[7]: left = 2
right = 7
%timeit -r 5 -n 1000000 left + right # 1 digit
left = 22
right = 77
%timeit -r 5 -n 1000000 left + right # 2 digits
left = 2222
right = 7777
%timeit -r 5 -n 10000 left + right # 4 digits
left = 22222222
right = 77777777
%timeit -r 5 -n 10000 left + right # 8 digits
left = 2222222222222222
right = 7777777777777777
%timeit -r 5 -n 10000 left + right # 16 digits
left = 222222222222222222222222
right = 777777777777777777777777
%timeit -r 5 -n 10000 left + right # 32 digits
```

(continues on next page)

(continued from previous page)

Run-times remain similar but increase (without doubling) as the numbers get larger and go beyond 64-bits, which is consistent with most of the computation being done in hardware. As mentioned before, since we only use 64-bit integers in M269, the run-time will always be within a certain bound and hence we can treat addition as a constant-time operation.

You may occasionally observe the mean run-times *decrease* as the inputs increase. A likely reason is that some other processes were running and so the previous measurements took longer or varied more across runs (look at the reported standard deviation) than they normally would.

Let's now look at exponentiation. We want to check if it's linear in the value of the exponent, so we must double the value, not the number of digits. Doubling the exponent quickly leads to multiplying integers that don't fit in 64 bits, but I'll do it anyway to see how the run-times increase.

```
[8]: base = 5
%timeit -r 5 -n 10000 base ** 1
%timeit -r 5 -n 10000 base ** 2
%timeit -r 5 -n 10000 base ** 4
%timeit -r 5 -n 10000 base ** 8
%timeit -r 5 -n 10000 base ** 16
%timeit -r 5 -n 10000 base ** 32
%timeit -r 5 -n 10000 base ** 64
```

17.6 ns ± 0.0339 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)

17.4 ns ± 0.0421 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)

(continues on next page)

(continued from previous page)

```
28.8 ns ± 0.0345 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
39.3 ns ± 0.193 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
52.2 ns ± 1.43 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
75.7 ns ± 0.178 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
111 ns ± 2.5 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
```

Overall, the run-time keeps increasing, but it's not doubling whenever the exponent doubles. The interpreter or the hardware is using a more efficient algorithm than repeatedly multiplying the base with itself. (You will see such an algorithm in Chapter 13.)



Note: To see the growth of run-times, keep doubling the input values or input sizes.

The fact that exponentiation has a lower complexity than linear in my computational environment (and yours too, probably) doesn't change our assumption. When analysing algorithms involving exponentiation, we shall still assume it's linear. The whole point of making these assumptions is to guarantee that you and your 700+ fellow M269 students obtain the same complexity for the same algorithm, and not different ones, depending on everyone's computational environment. You can imagine the confusion this would cause in forums, tutorials and TMAs.

A final note: in general, measuring the run-time on small inputs isn't very useful. First, the computational environment may make some optimisations for small values, which means the timings won't fit a clear growth rate. Second, even without optimisations the run-times may be so short that they will all be very similar, and look like constant run-time, or may fluctuate due to measuring errors. Third, we're interested in how algorithms cope with large, not small, inputs.



Note: In general, don't measure run-times for very small inputs.

Exercise 2.8.1

When measuring the run-times of exponentiation, would `base = 1` be a good choice? Explain why or why not.

Hint Answer

Exercise 2.8.2

To practice measuring run-times and editing notebooks, choose an operation (other than addition and exponentiation) from Section 2.2 and check whether its run-time is linear as the value or size of the operands increases. Add a code cell below this paragraph for your experiment.

Use the same number of runs and loops for each measurement, but different from what I used for addition. Since arithmetic operations take nanoseconds, choose at least one million loops to make the measurement relatively accurate. Since Python may optimise operations for values that fit in one byte, start the experiment with values greater than 255.

Hint Answer

2.9 Summary

This section summarises the previous ones. All definitions in this and future chapters are for the purposes of M269 only. Other texts and programming languages may define concepts and operations differently.

2.9.1 Python

A **module** is a code library. The statement `import m` imports the module `m`, making its contents available. To access a variable `v` or function `f` defined in `m`, write `m.v` or `m.f`.

An **identifier** is a name. It starts with a letter, followed by zero or more letters, underscores, periods or digits. The convention is to write variable and function names in lowercase, with underscores separating words.

A **keyword** is a word with a reserved meaning in Python. Keywords include `def`, `import`, `return` and can't be used as identifiers.

A **comment** starts with a hash symbol and goes until the end of the line. The Python interpreter ignores comments.

2.9.2 Data types

A **data type** (or just **type**) is a collection of values and operations on those values. An **abstract data type (ADT)** is a data type defined independently of any implementation.

Real numbers and floats

The **real numbers** are those that can be represented in decimal form, possibly with an infinite number of digits after the decimal point. The **positive** numbers are greater than zero; the **negative** numbers are less than zero.

The real number ADT consists of the real numbers and the operations in the next table. Python's `float` type implements an approximation of the real number ADT. The **floating-point numbers** (or **floats**) are the real numbers that can be represented with a fixed number of binary digits.

Operation	Mathematics	Python
negation	$-x$	<code>-x</code>
addition	$x + y$	<code>x + y</code>
subtraction	$x - y$	<code>x - y</code>
multiplication	$x \times y$	<code>x * y</code>
division	x / y	<code>x / y</code>
floor (round down)	$\text{floor}(x)$	<code>math.floor(x)</code>
maximum (largest)	$\max(x, y, \dots)$	<code>max(x, y, ...)</code>
minimum (smallest)	$\min(x, y, \dots)$	<code>min(x, y, ...)</code>

An **operator** is a symbol used to represent an operation. The **operands** are the values to which the operator applies. The right operand of the division operation can't be zero.

A **literal** is a direct representation of a value in Python. Integer and float literals start with a digit, followed by zero or more digits. A float literal includes a single period.

The real number π is approximated by the float `math.pi`.

Integers

The **integers** are the numbers $0, 1, -1, 2, -2, \dots$. Every integer is a real number. The **natural numbers** are the non-negative integers.

The integer ADT consists of the integers, the operations above and the operations in the next table. All operations, when applied to integers, produce integers, except division. Python's `int` type implements the integer ADT.

Operation	Mathematics	Python	Preconditions
floor division	$\text{floor}(x / y)$	<code>x // y</code>	$y \neq 0$
modulo	$x \bmod y$	<code>x % y</code>	$y \neq 0$
exponentiation or power	x^y	<code>x ** y</code>	$y \geq 0$

The **modulo** is the remainder of floor division. If $x \bmod y = 0$, then x is said to be a **multiple** of (or **divisible by**) y , and y is said to be a **factor** or **divisor** of x . An **even** integer is divisible by 2; an **odd** integer isn't.

Expressions

An **expression** is a single value or variable, or an operation applied to subexpressions. Operations are carried out in the following **precedence order**:

1. bracketed expressions, including functions
2. exponentiation
3. negation
4. multiplication, division, floor division, modulo

5. addition, subtraction.

Bracketed expressions are executed from the most to the least nested. Those at the same nesting level are executed from left to right. Operations with the same precedence are **left-associative** (executed from left to right), except negation and exponentiation, which are **right-associative** (executed right to left).

In Python, the evaluation of an expression may raise an error and stop, depending on the issue (division by zero, unknown name, or wrong syntax).

2.9.3 Functions

Most operations in M269, including the above, are **functions** in the mathematical sense: they produce a single output and don't modify the inputs. Functions are defined with the following template.

Function: the name of the function

Inputs: the name and type of each input variable

Preconditions: any conditions on the inputs

Output: the name and type of the output variable

Postconditions: how the output relates to the inputs

Preconditions state what must be true *before* we can apply the function to the input values; **postconditions** state what will be true *after* applying it. A function defined as:

Function: function name

Inputs: *input1*, a type1; *input2*, a type2; ...

Preconditions: conditions on *input1*, *input2*, ...

Output: *output*, an output type

Postconditions: conditions on *output*, *input1*, *input2*, ...

is implemented in Python as:

```
def function_name(input1: type1, input2: type2, ...) -> output_type:  
    """Prescribe what the function computes.  
  
    Preconditions: conditions on input1, input2, ...  
    Postconditions: conditions on output, input1, input2, ...  
    """  
    algorithm  
    return output
```

A **docstring** documents what a Python function does. It comes after the function header and starts and ends with three double quotes. Python's function `help` takes a function name and displays its header and docstring.

Python's function `print(expression1, expression2, ...)` displays the values of the given expressions.

An **assignment** statement is of the form ‘let *variable* be *expression*’ (in English) or `variable = expression` (in Python). The right-hand side expression is evaluated and its value assigned to the name. The assignment forms `x += y`, `x -= y`, etc., are abbreviations of `x = x + y`, `x = x - y`, etc.

An **algorithm** is a step-by-step procedure expressed as a structured list of unambiguous instructions. In this chapter, an algorithm is a sequence of assignments. Each assignment’s expression can only involve the input variables or the variables occurring on the left-hand side of previous assignments. At least one assignment must have the output variable on the left-hand side.

An algorithm implementing an operation is **correct** if for all inputs that satisfy the preconditions it produces an output satisfying the postconditions. If the input values don’t satisfy the preconditions, there’s no meaningful output and the algorithm may behave in any way.

Two consecutive Python lines of the form

```
output = expression
return output
```

can be simplified to `return expression`.

2.9.4 Complexity

The run-time of a statement or expression is measured with the IPython command `%timeit`. It runs the code many times to improve the measurement accuracy and takes the mean time. To set the exact number of repetitions, write

```
%timeit -r R -n N code to execute
```

where `R` and `N` are positive integer literals indicating the number of runs and the number of iterations per run, respectively.

The **complexity** of an algorithm is an indication of how its run-time grows for ever larger inputs.

Big-Theta notation $\Theta(e)$ states that, for large inputs, the run-time of an algorithm is proportional to the value of integer expression e based on the inputs’ values and sizes. The size of number n , written $|n|$, is how many digits it has.

An algorithm has **constant complexity**, written $\Theta(1)$, if its run-time doesn’t grow as the inputs get larger. An algorithm has **linear complexity** if its run-time grows proportionally to the inputs’ values or sizes. When the inputs’ value or size doubles, the run-time of a constant- or linear-time algorithm stays the same or doubles, respectively.

Although Python supports arbitrarily large integers, we will only use 64-bit integers. We can thus assume all arithmetic operations to have complexity $\Theta(1)$, except x^y , which has complexity $\Theta(y)$. The assignment and return statements also take constant time, after their expressions have been evaluated.

An algorithm that executes a fixed number of operations, all of them with constant complexity, has constant complexity too.

CHAPTER 3

BOOLEANS AND SELECTION

There are various ‘styles’ in which algorithms can be expressed. In the procedural style used in M269, algorithms are made of instructions put together in three ways: a **sequence** puts instructions one after the other; **selection** puts together alternative sets of instructions, allowing the execution of one set or the other; **iteration** repeats a set of instructions. Algorithms in the previous chapter were sequences of one or more assignments. This chapter covers selection and the next chapter covers iteration.

The selection of which set of instructions to execute is based on conditions, expressed as Boolean expressions. Hence this chapter also covers the Booleans, named after mathematician George Boole. Finally, the chapter introduces decision and classification problems, two large classes of problems that are solved using Booleans and selection.

This chapter supports the following learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter covers logical data and algorithms with choice
- Develop and apply algorithms and data structures to solve computational problems – this chapter introduces decision and classification problems.
- Analyse the complexity of algorithms to support software design choices – this chapter introduces best- and worst-case complexities.
- Write readable, tested, documented and efficient Python code – this chapter introduces testing in a more systematic way.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

3.1 Booleans

Besides numbers, most algorithms need logical values in order to represent binary properties (this section) and conditions ([Section 3.3](#)).

3.1.1 The Boolean ADT

The Boolean ADT consists of two logical values, true and false, and three logical operations, all of them functions in the mathematical sense (they produce one output, without changing the inputs):

- The output of the **conjunction** operation, written *left* and *right*, where *left* and *right* are Boolean values, is true if both operands are true, otherwise it's false.
- The output of the **disjunction** operation, written *left* or *right*, is false if both operands are false, otherwise it's true.
- The output of the **negation** operation, written not *right*, is true if the operand is false, and false if the operand is true.

We use the terms ‘arithmetic negation’ and ‘logical negation’ when it’s not clear from the context what ‘negation’ refers to.



Info: The conjunction, disjunction and negation operators are also written as \wedge , \vee and \neg , respectively.

These definitions can be written using the M269 function template. Here’s an example:

Function: conjunction

Inputs: *left*, a Boolean; *right*, a Boolean

Preconditions: true

Output: *result*, a Boolean

Postconditions: if $\text{left} = \text{right} = \text{true}$, then $\text{result} = \text{true}$, otherwise $\text{result} = \text{false}$

Only the input values that make the preconditions true are allowed. So, if the precondition is always true, any input value satisfies it. We henceforth write ‘true’ as the precondition when all input values are valid.

There are only two Boolean values, so if the output isn’t true, it must be false, and vice versa. This leads to a more succinct way of writing postconditions for operations with a Boolean output.

Postconditions: $\text{result} = \text{true}$ if and only if $\text{left} = \text{right} = \text{true}$

This states that the output is true if both inputs are true, and only in that case is the output true, i.e. for other input values the output is false.



Info: Some authors shorten ‘if and only if’ to ‘iff’.

An alternative way of defining the Boolean operations is to write a **truth table** that shows the outputs for all possible inputs.

<i>left</i>	<i>right</i>	<i>left and right</i>	<i>left or right</i>
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

<i>value</i>	<i>not value</i>
false	true
true	false



Info: TM112 Block 1 Section 1.3 introduces the logical operations and truth tables. Truth tables sometimes use 0 and 1, or F and T, for false and true.

Negation is right-associative and has higher precedence than conjunction, which in turn has higher precedence than disjunction. Conjunction and disjunction are left-associative. For example, the expression ‘true and not true or true’ is evaluated as follows:
 $(\text{true} \text{ and } (\text{not true})) \text{ or true} = (\text{true} \text{ and false}) \text{ or true} = \text{false or true} = \text{true}$.

Exercise 3.1.1

Fill the function definition template for the disjunction operation. Don’t forget to keep the backslash at the end of each line.

Function: disjunction

Inputs: the name and data type of each input

Preconditions: any conditions on the inputs

Output: the name and data type of the result

Postconditions: how the output relates to the inputs

Hint Answer

Exercise 3.1.2

Define the negation operation. Make the postcondition as brief as possible.

Function:

Inputs:

Preconditions:

Output:

Postconditions:

Hint Answer

3.1.2 Using Booleans

Booleans are used to represent properties that have only two possible states: on/off, open/closed, up/down, pass/fail, inside/outside, etc. We have to decide which state is represented by which Boolean value.

For example, a Boolean variable can represent whether a mobile phone is on or off. Let's assume that value true indicates that the phone is on. The variable name must reflect that choice, so I'll call it *phone on*. When bought, the phone is off, so the variable is initialised to false.

let *phone on* be false

If I prefer to represent the off state as the true value, then I must change the variable's name and initial value.

let *phone off* be true

A rule of thumb for choosing Boolean variable names is that they should become unambiguous yes/no questions when you add a question mark at the end. The name *phone is on* is a good one too, albeit more verbose, because 'phone is on?' is a precise yes/no question. The name *phone is* is a very poor one, because 'phone?' is an ambiguous question. It can mean 'do you need a phone?' or 'do you have a phone?' or something else.

Negation switches from one state to the other, like when flipping a switch or tapping a button. For example,

let *phone on* be not *phone on*

turns the phone off if it was on and vice versa.

Exercise 3.1.3

Is *phone state* a good name for a Boolean variable?

Answer

Exercise 3.1.4

Look at the settings of your mobile phone. List some that can be represented as Booleans.

Answer

3.1.3 The `bool` type

Python's `bool` type implements the Boolean ADT. The logical values true and false are represented by the literals `True` and `False`, which are keywords. Notice the initial uppercase letter!

The conjunction, disjunction and negation operations are written with the operators `and`, `or` and `not` respectively, which are keywords too. Here are some earlier examples, now in Python:

```
[1]: True and not True or True
```

```
[1]: True
```

```
[2]: phone_on = False
phone_on = not phone_on
phone_on
```

```
[2]: True
```

The Python interpreter stops evaluating conjunctions and disjunctions as soon as it can determine the value of the expression. If the left operand of a conjunction is false then the whole conjunction is false, so the right operand isn't evaluated. Likewise, if the left operand of a disjunction is true then the whole disjunction is true, so the right operand isn't evaluated. This is called the **short-circuit** evaluation of Boolean expressions. We'll see later why this is useful.

Exercise 3.1.5

Put parentheses in this expression so that it evaluates to false.

```
[3]: True and not True or True
```

I've removed the output in this and future exercises with Python code. Until you complete the exercises, their code output is necessarily wrong, so there's no point in showing it.

Hint Answer

Exercise 3.1.6

What may happen if you write Booleans in lowercase, i.e. `true` and `false`?

Hint Answer

3.1.4 Mistakes

In everyday English, ‘this or that’ often means ‘either this or that’, i.e. only one of them can be true, as in the statement ‘We will go to the movies or to the park’. In Boolean logic, disjunction never has that meaning. If it had, ‘true or true’ would be false.

It's easy to forget that conjunction is evaluated before disjunction, because it tends to be the other way round in English. The sentence ‘Alice is tall and has blue or brown eyes’ means that Alice is tall and that Alice has blue or brown eyes. But read as a Boolean expression, the sentence means that Alice is tall and has blue eyes, or that Alice has brown eyes. If Alice is short and has brown eyes, the sentence is false according to the usual meaning in English, but true according to the logical meaning.

```
[4]: is_tall = False
blue_eyes = False
brown_eyes = True
is_tall and blue_eyes or brown_eyes
```

[4] : True

The informal English meaning of the statement is obtained with parentheses:

[5] : `is_tall and (blue_eyes or brown_eyes)`

[5] : False



Note: The meaning of conjunction and disjunction in Boolean logic is not the same as the meaning of ‘and’ and ‘or’ in everyday English.

The short-circuit evaluation of conjunction and disjunction can lead to errors that are hard to detect. For example, the expression

[6] : `is_tall and blue_eyes or brown_eyes`

[6] : True

is evaluated as before, even though the second variable is misspelled. (Did you spot it?) There’s no name error because the right operand isn’t evaluated.

Exercise 3.1.7

Show the step-by-step evaluation of the following expression, using the precedence and associativity rules.

true or not true and false or not false

Hint Answer

3.2 Decision problems

A problem that has only two possible outcomes is called a **decision problem**, like deciding whether an integer is even or odd, or is a natural number or not. The output of such problems is best represented as a Boolean.

This section shows the process of solving a computational problem, using a decision problem as an example.

3.2.1 Problem definition

Consider the following informal problem description:

Many mobile phone apps require internet access to work. Implement a function that apps can call to check if the phone has an internet connection, i.e. if it’s not in flight mode (which ceases all communications) and Wi-Fi or mobile data are on.

The first step towards solving this problem is to define the function. Decision problems should also have names that become yes/no questions when appending a question mark. Here are some options (other names are possible):

Function: is internet connected

Function: connected to internet

Function: internet connection

Next, I must think about the inputs. To determine if there's an internet connection, the function requires the current state of the flight mode, Wi-Fi connection and data connection. Each of those can be on or off. The operation therefore has three Boolean inputs.

Inputs: *in flight mode*, a Boolean; *wifi on*, a Boolean; *data on*, a Boolean

Remember that variable names are in lowercase and can't include hyphens, so it's *wifi on*, not *Wi-Fi on*.

I can't think of any excluded value combinations, so:

Preconditions: true

In M269, the output of a decision problem is always a Boolean.

Output: *internet on*, a Boolean

The postcondition is given by the problem statement above: the phone has an internet connection if it's not in flight mode and Wi-Fi or mobile data are on. This statement, in everyday English, is like the statement about Alice: it doesn't use 'and' and 'or' with the same precedence as Boolean logic does. I must therefore translate it to the unambiguous English version we use in function definitions and algorithms, which relies on a precise meaning, precedence and associativity for 'and', 'or' and every other operation.

Postconditions: *internet on* = true if and only if *in flight mode* = false and (*wifi on* = true or *data on* = true)

A condition of the form ' $b = \text{true}$ ', where b is a Boolean variable, is true when b is true and false when b is false. In other words, the condition has the same value as the variable and can thus be simplified to just b . Similarly, a condition of the form ' $b = \text{false}$ ' has the opposite value of b and so can be simplified to 'not b '. Here's the whole definition, with these simplifications:

Function: internet connection

Inputs: *in flight mode*, a Boolean; *wifi on*, a Boolean; *data on*, a Boolean

Preconditions: true

Output: *internet on*, a Boolean

Postconditions: *internet on* if and only if not *in flight mode* and (*wifi on* or *data on*)

3.2.2 Problem instances

The next step of the process is to think how we'll check our solution, when we have it.



Info: Defining tests before devising an algorithm is part of test-driven development.

The function definition describes the general problem. A **problem instance** is a concrete problem: a collection of values, one per input, that satisfy the preconditions. The problem instances are those for which the algorithm must produce the correct output, i.e. that satisfies the postconditions.

A **test case** is a problem instance and its expected output. Writing a **test table** with one test case per row helps us check that we didn't forget any input, pre- or postcondition and helps us check our algorithm, once we write it.

The table has one column per input and output, and one extra column to describe the test case. You should include problem instances for the **edge cases**, i.e. values that occur at the boundaries. Examples of edge cases are: the input's lowest and highest possible values; zero (the boundary between negative and positive integers); 1 (the lowest positive number); -1 (the highest negative number). You can only include edge cases that satisfy the preconditions, or else the edge case is not a problem instance.

This function has three inputs, each with two possible values, so there are only $2 \times 2 \times 2 = 8$ problem instances. I consider the most interesting ones and leave the rest as an exercise for you.

Case	<i>in flight mode</i>	<i>wifi on</i>	<i>data on</i>	<i>internet on</i>
all on	true	true	true	false
all off	false	false	false	false
Wi-Fi connection	false	true	false	true
data connection	false	false	true	true
both connections	false	true	true	true

The chosen instances cover all cases for Wi-Fi and data: both, neither, or only one of them is on. In the first instance, the flight mode overrides the Wi-Fi and data connections and so there's no internet access.



Info: TM112 Block 1 Section 4.2.2 introduces test tables.

Exercise 3.2.1

Write the three remaining problem instances in the table below.

Case	<i>in flight mode</i>	<i>wifi on</i>	<i>data on</i>	<i>internet on</i>

Hint Answer

3.2.3 Algorithm

The next step is to write an algorithm in plain but unambiguous English (and some maths, if necessary). This is the most creative step of the whole process: there's no recipe for it. M269 teaches several algorithmic techniques to put you in the right direction, but coming up with the algorithm can still be hard work.

For this problem I must translate the postcondition into an assignment:

let *internet on* be (not *flight mode*) and (*wifi on* or *data on*)

The first pair of parentheses is redundant because negations are evaluated before conjunctions, but I think they make the expression easier to read and understand.

3.2.4 Complexity

The next step is to analyse the complexity of our algorithm and, if it turns out to be too high, i.e. the algorithm is too inefficient, go back to the previous step and improve or completely redesign the algorithm.

My algorithm does a fixed number of operations: one assignment and three logical operations (one of each). The algorithm has constant complexity, assuming that logical operations have constant complexity. Do you think that's a reasonable assumption?

Definitely, for three reasons. First, logical operations are executed in hardware, by the computer's arithmetic and logic unit (ALU). Second, Booleans can't grow in value and size, unlike numbers, because there's only two of them. Hence the run-time of logical operations can't grow. Third, computing the result of a logical operation consists of a fixed number of elementary steps to look up a truth table of at most four rows.



Info: TM112 Block 1 Section 3.1.1 introduces the ALU.

The algorithm has complexity $\Theta(1)$. That's as good as it gets.

3.2.5 Code and tests

Finally, we translate the function definition and the algorithm to a Python function. The header is rather long and so I write the parameters over multiple lines.

```
[1]: def internet_connection(in_flight_mode: bool, wifi_on: bool, data_on: bool) -> bool:  
    """Return whether there's a connection to the internet.  
  
    Postconditions: the output is true if and only if  
    (not in_flight_mode) and (wifi_on or data_on)  
    """  
    internet_on = (not in_flight_mode) and (wifi_on or data_on)  
    return internet_on
```

To check the algorithm and its translation to Python, I must call the Python function for each test case in my table and compare the interpreter's output with the last column of the table.

```
[2]: internet_connection(True, True, True)  
[2]: False  
  
[3]: internet_connection(False, False, False)  
[3]: False  
  
[4]: internet_connection(False, True, False)  
[4]: True  
  
[5]: internet_connection(False, False, True)  
[5]: True  
  
[6]: internet_connection(False, True, True)  
[6]: True
```

Exercise 3.2.2

Add the calls for the remaining three problem instances. The quickest way is to copy and paste code cells and change as necessary.

Answer

3.2.6 Performance

The last step is to measure the run-time of the implemented function. As explained in the previous chapter, complexity analysis only gives us an idea of how the run-time grows for increasingly large inputs: it doesn't tell us whether the implemented algorithm runs slow or fast.

Often we won't do any run-time measurement as there's little point in it, like in this example. I expect the run-times to be pretty small, given that the function just does a few logical operations, one assignment and one return statement (which also takes constant time).

Let's check that logical operations take constant time.

```
[7]: %timeit -r 3 -n 1000 internet_connection(True, True, True)
%timeit -r 3 -n 1000 internet_connection(False, False, False)

52.5 ns ± 0.472 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
˓→each)
61.8 ns ± 0.465 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
˓→each)
```

On my computer the second call takes longer than the first one. Can you explain the reason?

The first call doesn't execute the disjunction operation. It short-circuits the conjunction because `in_flight_mode` is true and thus the left operand (`not in_flight_mode`) is false. The second call takes longer to run because it doesn't short-circuit any operation: the left operand of the conjunction is true and the left operand of the disjunction (`wifi_on`) is false.

To compare like for like, we must make two calls that do the same number of evaluations. Here are two that don't short-circuit any expression and therefore execute all logical operations (negation, conjunction and disjunction).

```
[8]: %timeit -r 3 -n 1000 internet_connection(False, False, False)
%timeit -r 3 -n 1000 internet_connection(False, False, True)

61.8 ns ± 0.425 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
˓→each)
61.8 ns ± 0.343 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
˓→each)
```

The run-times are now more similar.

Exercise 3.2.3

Measure the run-times of two function calls that don't short-circuit the conjunction, but short-circuit the disjunction.

```
[9]: # replace with your code
```

Hint Answer

3.3 Boolean expressions

Besides two-state properties, Booleans can represent whether a condition holds or not, e.g. whether a temperature is below some threshold. To express such conditions we need to be able to compare values.

3.3.1 Comparisons

There are six comparison operations.

The **equality** operation checks if two expressions have the same value. Any values can be compared for equality, not just Booleans. The equality operator is `=` in mathematics and `==` in Python.

```
[1]: 3 == 5
```

```
[1]: False
```

```
[2]: True == 42 # any values can be compared
```

```
[2]: False
```

```
[3]: False == False
```

```
[3]: True
```

The **inequality** operation, written \neq in maths and `!=` in Python, checks if two values are different. It returns the opposite value of equality.

```
[4]: 3 != 5
```

```
[4]: True
```

```
[5]: True != 42 # any values can be compared
```

```
[5]: True
```

```
[6]: False != False
```

```
[6]: False
```

The remaining comparison operators are $<$, \leq , $>$, \geq in mathematics, and `<`, `<=`, `>`, `>=` in Python. They can be applied to various data types, as we shall see. For the moment, they only apply to numbers.

```
[7]: 4 >= 3.5
```

```
[7]: True
```

The comparison operators have lower precedence than arithmetic operators, but higher precedence than logical operators. This allows writing without parentheses complex Boolean expressions with all three kinds of operations.

```
[8]: 51 - 20 == 29 + 2 and not False
```

```
[8]: True
```

Can you explain how the result was obtained?

The arithmetic subexpressions are evaluated first: $31 = 31$ and not false. The comparisons are evaluated next: true and not false. Finally, the logical operators are applied, leading to the final true value.

Comparisons are left-associative. A double inequality like $1 \leq \text{day} \leq 31$, which checks whether a variable's value is within a certain interval, is equivalent to ' $1 \leq \text{day}$ and $\text{day} \leq 31$ '. The latter doesn't need parentheses because comparisons are evaluated before logical operations. The expression $0 < \text{day} < 32$ is equivalent if day is an integer, but less clear in my opinion, because it doesn't state the lower and upper bounds of the interval explicitly. Python supports the double inequality notation, as shown in the next example.



Info: MU123 Unit 2 Section 4.2 and MST124 Unit 3 Section 1.1 introduce double inequalities and intervals.

The short-circuit conjunction and disjunction are useful to guard against errors. For example, let's suppose I want to check if a fraction corresponds to a probability, i.e. is in the interval from zero to one. If the denominator is zero, the fraction is undefined and thus not a probability.

```
[9]: numerator = 5
denominator = 0
is_probability = denominator != 0 and 0 <= numerator / denominator
               <= 1
is_probability
[9]: False
```

There's no division by zero error. The left operand of the conjunction guarantees that the right operand is only evaluated if the fraction has a defined value. If you wish, you can change the denominator to, say, one ($\frac{5}{1}$ isn't a probability) and then ten ($\frac{5}{10}$ is a chance of one in two).

We will avoid writing algorithms that depend on the short-circuit behaviour, because not all programming languages implement it.

We assume comparisons are done in hardware and take constant time.

```
[10]: %timeit -r 3 -n 1000 9 > 8
%timeit -r 3 -n 1000 9_999_999 > 9_999_998
%timeit -r 3 -n 1000 9 != False
%timeit -r 3 -n 1000 9_999_999 != False

14.8 ns ± 0.257 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
16 ns ± 0.325 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
22.3 ns ± 0.477 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
22.2 ns ± 0.118 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
```

3.3.2 Mistakes

In Python, a single equals sign is the assignment operator, not equality! The Boolean expression `x == y` checks if `x` and `y` refer to the same value, whereas the assignment `x = y` makes `x` refer to the same value as `y`. For example, `phone_on == False` checks if the phone is off, whereas `phone_on = False` turns it off. The Python interpreter may not raise an error and execute the assignment, even if you intended a comparison for equality. The interpreter can't read your mind (fortunately!) and executes any program that looks meaningful.



Note: The difference between what we think we wrote and what we actually wrote is a source of many subtle errors. Always double-check your code.

More often than not, Python can detect the error, like in this case:

```
[11]: True = 42
       Cell In[11], line 1
             True = 42
             ^
SyntaxError: cannot assign to True
```

In `!=`, `<=` and `>=` there must be no space before the equals sign. If there is one, it separates a single operation (e.g. `>=`) into two (`>` and assignment), which leads to a syntax error.

```
[12]: 4 > = 3
       Cell In[12], line 1
             4 > = 3
             ^
SyntaxError: invalid syntax
```

Another kind of mistake is to compare floats for equality. Due to their approximate representation of most decimal numbers, comparisons for equality may not work.

```
[13]: 0.1 + 0.2 == 0.3
[13]: False
```

One way to handle such cases is to check if the value of the expression is within some error margin.

```
[14]: margin = 0.001
       0.3 - margin <= 0.1 + 0.2 <= 0.3 + margin
[14]: True
```

Comparisons having higher precedence than logical operations occasionally leads to somewhat

surprising mistakes. For example, `False == not True` looks like an innocuous expression showing that false is the opposite of true. However, when evaluating it...

```
[15]: False == not True
```

```
Cell In[15], line 1
  False == not True
          ^
SyntaxError: invalid syntax
```

the interpreter raises a syntax error because it evaluates the comparison `False == not` first, which makes no sense. We must write `False == (not True)` to evaluate the expression as intended.

We're used to arithmetic expressions since childhood, but the precedence rules of Boolean expressions are unfamiliar to most of us. Forgetting brackets can lead to syntax errors or, even worse, the wrong value being computed, e.g. due to the precedence of conjunction over disjunction. I therefore recommend:



Note: In a Boolean expression, put parentheses around each separate condition.

This may lead to a Boolean expression with redundant brackets, but clarity is worth a few extra characters.

Exercise 3.3.1

Write an expression, in mathematical notation, not Python, that captures the meaning of 'either *left* or *right*', where *left* and *right* are Boolean values. The expression is true if and only if exactly one of *left* and *right* is true.

Answer

Exercise 3.3.2

1. List what's wrong in the expression `-5 > = value > -20` but `value not 0`.
2. Resolve the errors to get a syntactically correct expression.
3. Simplify the expression.

Hint Answer

3.4 Classification problems

So far, all algorithms have been just sequences of instructions: first do this, then do this, next do that, etc. Most algorithms don't execute all their instructions, only some of them, based on certain conditions. Programming languages have conditional statements that allow us to select

which instructions to execute under which conditions. Let's consider a classic problem that requires selection and conditions, stated as Boolean expressions. I follow again the same process to solve it.



Info: TM112 introduces selection in Block 1 Sections 4.3 and 4.4, and includes a more complicated version of the following problem in Block 2 Section 2.5.

3.4.1 Problem definition and instances

Given a mark (an integer from 0 to 100), we wish to award the corresponding pass grade, from 1 (distinction) to 5 (fail). This is a **classification problem**: each of the many possible input values is classified into one of a few categories. A decision problem is a classification problem with only two categories.



Info: Question 19 of the TM112 Block 1 Quiz is a classification problem too: given the Richter magnitude of an earthquake, classify it as a minor, moderate or major earthquake. TM358 introduces machine learning algorithms for difficult classification problems.

Let's assume that, in some fictitious module, the pass grade boundaries are 40, 50, 60 and 80. Marks on the boundaries are awarded the higher pass grade.

Function: grading

Inputs: $mark$, an integer

Preconditions: $0 \leq mark \leq 100$

Output: $pass$, an integer

Postconditions:

- $pass = 5$ if $0 \leq mark < 40$
- $pass = 4$ if $40 \leq mark < 50$
- $pass = 3$ if $50 \leq mark < 60$
- $pass = 2$ if $60 \leq mark < 80$
- $pass = 1$ if $80 \leq mark \leq 100$

There's no simple formula to transform a mark into a pass grade. It's easier to write one condition per grade.



Info: Functions that have different formulas for different intervals of the input values are called piecewise-defined functions in MST124 Unit 3 Section 1.3.

It's easy to make a test table because...



Note: The edge cases for a classification problem are the categories' boundaries.

Case	mark	pass
lowest fail	0	5
highest fail	39	5
lowest pass 4	40	4
highest pass 4	49	4
lowest pass 3	50	3
highest pass 3	59	3
lowest merit	60	2
highest merit	79	2
lowest distinction	80	1
highest distinction	100	1

3.4.2 Algorithm

The algorithm for a classification problem is a sequence of statements of the form ‘if the input value is like this, then the category is that one’, which is essentially how the postconditions are expressed, making their translation to an algorithm rather easy.

1. if $0 \leq \text{mark} < 40$:
 1. let pass be 5
2. if $40 \leq \text{mark} < 50$:
 1. let pass be 4
3. if $50 \leq \text{mark} < 60$:
 1. let pass be 3
4. if $60 \leq \text{mark} < 80$:
 1. let pass be 2
5. if $80 \leq \text{mark} \leq 100$:
 1. let pass be 1

The algorithm simply follows the typographic convention in English of introducing sublists of items (here, instructions) with colons, and indenting them. We refer to individual steps as 1, 1.1, 2, 2.1, etc.

This is a fine algorithm. It's easy to check the algorithm is correct because it directly follows the postconditions, showing explicitly every category boundary. The conditions are mutually exclusive (at most one is true for each problem instance) and comprehensive (at least one is true

for each problem instance). For any input value that satisfies the preconditions, exactly one condition is true, and this allows the output to be uniquely determined.

However, the algorithm is not the most efficient. If the mark is, say, 30, only the first condition applies, but the remaining conditions are checked too, even though they're false. A more efficient algorithm stops as soon as the grade is determined.

1. if $0 \leq \text{mark} < 40$:
 1. let pass be 5
2. otherwise if $40 \leq \text{mark} < 50$:
 1. let pass be 4
3. otherwise if $50 \leq \text{mark} < 60$:
 1. let pass be 3
4. otherwise if $60 \leq \text{mark} < 80$:
 1. let pass be 2
5. otherwise if $80 \leq \text{mark} \leq 100$:
 1. let pass be 1

The word ‘otherwise’ indicates that the next condition is checked only if the previous ones failed. This allows us to check fewer conditions. For example, if the second condition gets evaluated, it means the first one is false, i.e. the mark isn't in the range 0–39. It's therefore redundant to check if it's greater or equal to 40 (because it is). The algorithm becomes:

1. if $0 \leq \text{mark} < 40$:
 1. let pass be 5
2. otherwise if $\text{mark} < 50$:
 1. let pass be 4
3. otherwise if $\text{mark} < 60$:
 1. let pass be 3
4. otherwise if $\text{mark} < 80$:
 1. let pass be 2
5. otherwise if $\text{mark} \leq 100$:
 1. let pass be 1

We can omit the first part of the first check and all of the final check because $0 \leq \text{mark} \leq 100$ is always true, due to the preconditions.

1. if $\text{mark} < 40$:
 1. let pass be 5
2. otherwise if $\text{mark} < 50$:

1. let *pass* be 4
3. otherwise if *mark* < 60:
 1. let *pass* be 3
4. otherwise if *mark* < 80:
 1. let *pass* be 2
5. otherwise:
 1. let *pass* be 1

The last part of the algorithm states that if none of the previous cases applies then the grade must be a distinction.



Note: Check input intervals from the lowest to the highest or from the highest to the lowest, so that you can simplify the conditions.

The following exercises show the importance of the order in which conditions are checked.

Exercise 3.4.1

My algorithm, repeated below, does one comparison for a pass 5 mark and four comparisons for a pass 1 mark.

1. if *mark* < 40:
 1. let *pass* be 5
2. otherwise if *mark* < 50:
 1. let *pass* be 4
3. otherwise if *mark* < 60:
 1. let *pass* be 3
4. otherwise if *mark* < 80:
 1. let *pass* be 2
5. otherwise:
 1. let *pass* be 1

Change the algorithm so that the minimum number of comparisons is made for pass 1 marks, and the maximum number of comparisons is made for pass 5 marks. Do as few comparisons as possible.

Hint Answer

Exercise 3.4.2

Consider the following grading algorithm. Is it correct?

1. if $60 \leq \text{mark} < 80$:
 1. let pass be 2
2. otherwise if $0 \leq \text{mark} < 40$:
 1. let pass be 5
3. otherwise if $50 \leq \text{mark} < 60$:
 1. let pass be 3
4. otherwise if $80 \leq \text{mark} \leq 100$:
 1. let pass be 1
5. otherwise if $40 \leq \text{mark} < 50$:
 1. let pass be 4

Hint Answer

Exercise 3.4.3

Consider the following simplification of the above conditions.

1. if $\text{mark} < 80$:
 1. let pass be 2
2. otherwise if $\text{mark} < 40$:
 1. let pass be 5
3. otherwise if $\text{mark} < 60$:
 1. let pass be 3
4. otherwise if $\text{mark} \leq 100$:
 1. let pass be 1
5. otherwise if $\text{mark} < 50$:
 1. let pass be 4

Explain why the algorithm is incorrect by showing **one counter-example**: a problem instance for which the algorithm produces the wrong output.

Copy the text cell with the algorithm to below this paragraph and fix it. You can only change the conditions, not their order.

Hint Answer

3.4.3 Complexity

The fourth version of the algorithm, the one just before the exercises, always does one assignment, but it may execute one to four comparisons, against 40, 50, 60 and 80 marks, depending on the input value. To handle these situations, complexity analysis distinguishes between best- and worst-case scenarios.

A **best-case scenario** is a group of problem instances that lead to the algorithm doing the least work, i.e. running fastest. For the algorithm at hand, a best-case scenario is a mark from 0 to 39, as only one comparison is made. A **worst-case scenario** is a group of problem instances that require the algorithm to do the most work, i.e. running slowest. A worst-case scenario for this algorithm is a mark from 60 upwards, because it requires four comparisons. (Best and worst cases are for the algorithm, not for the student's grade.)

The algorithm does one assignment and one comparison in the best-case scenario. That's a fixed number of constant-time operations, so the algorithm has **best-case complexity** $\Theta(1)$. In the worst-case scenario, the algorithm does one assignment and four comparisons, which takes constant time too. The **worst-case complexity** is also $\Theta(1)$. When an algorithm's best- and worst-case complexities are the same, we simply state the algorithm's complexity without any qualification. In this example, we just state that the algorithm has constant complexity.

As you'll see in later chapters, there may be different equivalent best-case (or worst-case) scenarios, so we tend to speak of *a* best- or worst-case scenario rather than *the* best- or worst-case scenario. All best- (or worst-) case scenarios necessarily have the same complexity, otherwise some scenarios would be better (respectively, worse) than others.

3.4.4 Code

Python's syntax closely follows English, with 'if', 'otherwise' and 'otherwise if' being the keywords `if`, `else` and `elif`, respectively.

Problem definitions indicate the output's name so that postconditions can refer to it. Function headers don't name the output, so I write instead 'the output is 4' or more simply 'return 4' in the docstring. I take the opportunity to rely on the preconditions, i.e. that marks go from 0 to 100, to slightly simplify the formulation of the postconditions.

```
[1]: def grading(mark: int) -> int:
    """Return the pass grade, from 1 to 5, for the given mark.

    Preconditions: 0 <= mark <= 100
    Postconditions:
        - if mark < 40, return 5
        - if 40 <= mark < 50, return 4
        - if 50 <= mark < 60, return 3
        - if 60 <= mark < 80, return 2
        - if mark >= 80, return 1
    """
    if mark < 40:
        grade = 5
    elif mark < 50:
        grade = 4
    elif mark < 60:
        grade = 3
    elif mark < 80:
        grade = 2
    else:
        grade = 1
```

(continues on next page)

(continued from previous page)

```
    grade = 4
elif mark < 60:
    grade = 3
elif mark < 80:
    grade = 2
else:
    grade = 1
return grade
```

We can immediately return the grade once it's determined, which leads to another version:

```
if mark < 40:
    return 5
if mark < 50:
    return 4
if mark < 60:
    return 3
if mark < 80:
    return 2
return 1
```

In English, we write this algorithm as:

1. if mark < 40:
 1. let *pass* be 5
 2. stop
2. if mark < 50:
 1. let *pass* be 4
 2. stop
3. if mark < 60:
 1. let *pass* be 3
 2. stop
4. if mark < 80:
 1. let *pass* be 2
 2. stop
5. let *pass* be 1

With the explicit ‘stop’ instruction we don’t need to use ‘otherwise’. For this example, I prefer to use ‘otherwise’ instead of ‘stop’, as it makes the algorithm shorter and easier to understand, in my view.

Remember that our algorithms in English must assign a value to the output variable mentioned in the function definition template, whereas algorithms in Python can directly return the value, as there's no output name in the Python function header.

Some authors advocate having a single stopping point in an algorithm, usually implicit after the final step, as having several may make the algorithm harder to understand. In M269 you can use either 'style': one style may be more convenient for the algorithm you're working on.

3.4.5 Tests

Previously, we called the function on each problem instance and manually checked if the output was the same as in the last column of the test table. We can use Boolean expressions to compare the returned grade against the expected one. It's much faster and less error-prone to see if all outputs are true than to check if each output is the right grade.

```
[2]: grading(0) == 5
```

```
[2]: True
```

```
[3]: grading(39) == 5
```

```
[3]: True
```

```
[4]: grading(40) == 4
```

```
[4]: True
```

```
[5]: grading(49) == 4
```

```
[5]: True
```

```
[6]: grading(50) == 3
```

```
[6]: True
```

```
[7]: grading(59) == 3
```

```
[7]: True
```

```
[8]: grading(60) == 2
```

```
[8]: True
```

```
[9]: grading(79) == 2
```

```
[9]: True
```

```
[10]: grading(80) == 1
```

```
[10]: True
```

```
[11]: grading(100) == 1
```

```
[11]: True
```

3.4.6 Performance

Remember that in the best-case scenario (any fail mark) only one comparison is made, whereas in the worst-case scenario (any distinction mark) four comparisons are made. For curiosity's sake, let's see the difference between the best- and the worst-case run-times.

```
[12]: %timeit -r 3 -n 1000 grading(0)
%timeit -r 3 -n 1000 grading(100)
```

```
49.3 ns ± 0.367 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
```

```
→ each)
```

```
73.2 ns ± 0.484 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
```

```
→ each)
```

On my computer the second call doesn't take four times longer to run, because most of the time goes into calling the function and returning from it, not in executing comparisons.

3.4.7 Mistakes

Writing `else if` instead of `elif` is a syntax error.

If the conditions are not comprehensive, i.e. don't cover all possible input values, then no output is computed for some problem instances.

If the conditions are not mutually exclusive, i.e. they overlap for some input values, then some problem instances can be classified in more than one category. The algorithm will assign the category for the first condition that succeeds, so the order in which conditions are checked may lead to the correct answer for some inputs, but not for others. Consider again the algorithm of Exercise 3.4.3:

1. if $mark < 80$:
 1. let $pass$ be 2
2. otherwise if $mark < 40$:
 1. let $pass$ be 5
3. otherwise if $mark < 60$:
 1. let $pass$ be 3
4. otherwise if $mark \leq 100$:
 1. let $pass$ be 1
5. otherwise if $mark < 50$:
 1. let $pass$ be 4

The conditions overlap, e.g. marks up to 40 satisfy all conditions, and due to the order they're checked, marks below 60 get the wrong grade.



Note: Write at least one test for each category, so that you're more likely to catch missing and overlapping conditions.

3.5 Practice

This section allows you to practise the problem-solving process using Boolean expressions and selection statements.

3.5.1 Phone calls

Assume that a mobile phone can make and receive calls if flight mode is off and if the signal strength (indicated by zero to four vertical bars on the phone's screen) is at least two. Write a function that checks if a phone can make and receive calls. The following exercises break down the process.

Exercise 3.5.1

1. What kind of problem is this?
2. Fill in the template.

Function:

Inputs:

Preconditions:

Output:

Postconditions:

Hint Answer

Exercise 3.5.2

Write a test table. Change the generic column headings. Add columns and rows as needed.

Case	input	output

Hint Answer

Exercise 3.5.3

Write the algorithm.

Hint Answer

Exercise 3.5.4

1. Translate the function definition and algorithm to a Python function.

```
[1]: # def ...
```

2. Run the test cases in your test table. Add code cells as necessary.

```
[2]: # replace this with your code
```

Hint Answer

3.5.2 Maximum

If there weren't a function to calculate the maximum of several numbers, we could define our own, with only two inputs, and call it repeatedly. For example, `maximum(x, maximum(y, z))` calculates the largest of three values, two at a time.

Note that the maximum problem, although it requires selection to be solved, is not a classification or decision problem.

Exercise 3.5.5

1. Define a maximum function with only two inputs.

Function: `maximum`

Inputs:

Preconditions:

Output:

Postconditions:

2. Write a test table.

Hint Answer

Exercise 3.5.6

Write the algorithm.

Answer

Exercise 3.5.7

1. Translate the function definition and algorithm to a Python function.

```
[3]: # replace this with your code
```

2. Translate your test table to Python code. Add code cells as necessary.

```
[4]: # replace this with your code
```

*Answer***3.5.3 Leap year**

In the Gregorian calendar introduced in October 1582 and used by many countries, leap years have an extra day (29 February). Every year that is divisible by four is a leap year, except for those that are divisible by 100. However, years divisible by 100 are leap years if they're divisible by 400. For example, 1600 and 2000 were leap years, but 1700, 1800 and 1900 weren't.

Exercise 3.5.8

1. Define a function that decides if a given year is a leap year.

Function:**Inputs:****Preconditions:****Output:****Postconditions:**

2. Change the column names of this test table and add cases.

Case	input	output
divisible by 400	1600	true
divisible by 100 but not by 400	1700	false

*Hint Answer***Exercise 3.5.9**

Write an algorithm.

*Hint Answer***Exercise 3.5.10**

1. Translate the function definition and algorithm to Python.

```
[5]: # replace this with your code
```

2. Translate your test table to Python code. Add code cells as necessary.

```
[6]: # replace this with the test for 1600
```

```
[7]: # replace this with the test for 1700
```

Answer

3.6 Summary

3.6.1 Booleans

The **Boolean** ADT and Python's `bool` type consist of two values and three logical operations.

Values and operations	Boolean ADT	<code>bool</code> type
values	true false	True False
negation	not <i>value</i>	not <i>b</i>
conjunction	<i>left</i> and <i>right</i>	<i>left</i> and <i>right</i>
disjunction	<i>left</i> or <i>right</i>	<i>left</i> or <i>right</i>

In Python, the Boolean literals and logical operators are keywords.

A **negation** is true if and only if the operand is false. A **conjunction** is true if and only if both operands are true. A **disjunction** is true if and only if at least one of the operands is true. In Python, conjunction and disjunction are **short-circuit operators**: the right operand isn't evaluated if the left operand determines the result.

The comparison operations compare any two values *v1* and *v2*, or any two numbers *n1* and *n2*, and produce a Boolean:

Operation	Maths	Python
equality	$v1 = v2$	<code>v1 == v2</code>
inequality	$v1 \neq v2$	<code>v1 != v2</code>
less than	$n1 < n2$	<code>n1 < n2</code>
less than or equal to	$n1 \leq n2$	<code>n1 <= n2</code>
greater than	$n1 > n2$	<code>n1 > n2</code>
greater than or equal to	$n1 \geq n2$	<code>n1 >= n2</code>

All logical and comparison operators are left-associative, except negation, which is right-associative. The precedence of operations, from highest to lowest, is: arithmetic operations, comparisons, negations, conjunctions, disjunctions.

All comparison and logical operations take constant time.

3.6.2 Solving problems

The M269 problem-solving process has the following steps.

1. Define the function by filling in the template given in the previous chapter.
2. Write a test table. Each row is a **test case**: a **problem instance** (input values that satisfy the preconditions) and its corresponding output. The table has one column per input, one column for the output, and one column to describe the test case. The test table should ideally cover all edge cases (boundary values).
3. Write an algorithm in plain English with mathematical notation as necessary. To check if the algorithm is incorrect, try to find a **counter-example** – a problem instance that leads to a wrong output.
4. Analyse the complexity of the algorithm by looking at how many operations it executes and the complexity of each one. Check if there's a **best- or worst-case scenario**, i.e. a collection of problem instances that require the algorithm to do the least (respectively, most) work. All best- or worst-case scenarios have respectively the same **best- or worst-case complexity**. If the best- and worst-case complexities are the same, just state ‘the algorithm has complexity ...’.
5. Translate the function definition and algorithm to a Python function.
6. Test the Python function by calling it with each problem instance in the test table and comparing the returned value to the expected output. The function passes the tests if all equalities are true.
7. Measure the function’s performance for a range of input values, with the `%timeit` command.

The process is iterative: doing one step may require revising a previous one.

3.6.3 Classification problems

A **classification problem** assigns each problem instance to a category. A **decision problem** is a classification problem with two categories, i.e. a problem with a yes/no answer, represented as a Boolean output.

A simple classification algorithm is a sequence of selection statements (also called conditional statements) to determine the category.

1. if *input* is:
 1. let *category* be ...
2. if *input* is:
 1. let *category* be ...
3. etc.

For this algorithm to be correct, the conditions must be mutually exclusive (they don't overlap) and comprehensive (they cover all problem instances). This ensures that for each valid input exactly one condition is true, i.e. each problem instance is classified in exactly one category, and that the order of the selection statements doesn't matter.

The condition of a statement starting with ‘otherwise’ is only evaluated if the previous ones failed. This may allow for simpler conditions, depending on their order. The last condition can be omitted.

1. if *input* is:
 1. let *category* be ...
 2. otherwise if *input* is:
 1. let *category* be ...
 3. etc.
 4. otherwise:
 1. let *category* be the remaining one

Instead of using ‘otherwise’, an algorithm in English may use the instruction ‘stop’ to immediately stop the execution of the algorithm:

1. if *input* is:
 1. let *category* be ...
 2. stop
2. if *input* is:
 1. let *category* be ...
 2. stop
3. etc.
4. let *category* be the remaining one

The words ‘if’, ‘otherwise if’ and ‘otherwise’ in the English algorithm are translated to the Python keywords `if`, `elif`, and `else`, respectively.

```
if input ....:
    category = ...
elif input ....:
    category = ...
...
else:
    category = ...
```

The two steps

1. let *output* be expression
2. stop

are translated to Python as `return expression`.

CHAPTER 4

SEQUENCES AND ITERATION

There's not much that algorithms using only sequences and selections of instructions can do. The ability to handle large inputs comes from iteration: the ability to repeatedly loop through instructions, either for a fixed number of times or until some condition occurs. The three ways of structuring instructions can be arbitrarily combined: we can have a sequence of selections, a selection of iterations, an iteration of iterations, etc. The power of computation comes from this flexibility. This chapter covers the loop instructions used in M269.

Likewise, there's not much that algorithms using only numeric and logical variables can do. The ability to handle large data comes from aggregating individual values into **collections**, sometimes also called **containers**. One of the simplest collection types is a sequence of data items, one after the other. Data sequences almost always require iterative algorithms in order to process all items, one by one. This chapter introduces the sequence ADT and four implementations of it in Python.

This chapter supports the following learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter introduces sequence data types and iterative algorithms.
- Analyse the complexity of algorithms to support software design choices – this chapter introduces the analysis of algorithms with loops.
- Write readable, tested, documented and efficient Python code – this chapter introduces a simple framework to automatically test functions.



Note: This is the longest chapter in the book. Don't feel rushed to complete it in one week. You can use part of next week to catch up.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

4.1 The Sequence ADT

A text is a sequence of characters. A to-do list is a sequence of tasks. A top 40 chart is a sequence of songs. A queue is a sequence of people. A **sequence** is a collection of items in a particular order: one item comes first, another comes second, and so on until the last item. The items in a sequence are also called its **members** or **elements**. An **empty sequence** has no elements, e.g. when all tasks have been done.



Info: In mathematics, sequences are usually lists of numbers that follow a certain pattern, like 1, 4, 7, 10, ... or 5, 10, 20, 40, ... They are special cases of the sequences used in computing. Numeric sequences are introduced in MU123 Unit 9 Section 1.1 and MST124 Unit 10.

In M269 we write sequences as comma-separated lists of items, enclosed in parentheses: () is the empty sequence, while (1, 2, true) is a sequence of two integers and one Boolean. The order matters, so (1, 2, 3) and (1, 3, 2) are different sequences.



Info: There's no standard bracket to enclose sequences. Some texts use { ... } or ⟨ ... ⟩ or no brackets at all, like the MU123 and MST124 books.

The following are the most common operations on **immutable** sequences, i.e. sequences that cannot be modified. Operations on **mutable** sequences are given in [Section 4.6](#).

4.1.1 Inspecting sequences

The following functions allow us to obtain some information about a sequence.

Size

The **length** or **size** of a sequence s , written $|s|$, is the number of its elements. The empty sequence has length zero. We assume the size is stored in memory together with the sequence, and that the size is computed and updated when the sequence is created and modified. Hence the length operation can look up the size in constant time.

Indexing

It's possible to obtain the first, second, ..., last item of a sequence with the **indexing** operation. In the sequence (true, false), true is at index zero and false is at index one. Because indices start at 0, the last index is one less than the sequence's length.

The indexing function takes a sequence and an index, and returns the item at that index. The members of a sequence can be of any type, so we need a general ADT that includes every possible data item. In M269 we call it the **object** ADT. It only has two operations: equality and inequality. We can now define the indexing operation.

Function: indexing

Inputs: $values$, a sequence; $index$, an integer

Preconditions: $0 \leq index < |values|$

Output: $value$, an object

Postconditions: $value$ is the n -th item of $values$, with $n = index + 1$

The indexing operation is written in mathematics as s_i , for a sequence s and index i . In computing, the more common notation is $s[i]$. You can use both notations in M269.

In M269, all data we need to process fits and is stored in the computer's main random-access memory (RAM). Any RAM position can be accessed in the same time, so we assume that indexing takes constant time.

Exercise 4.1.1

Does the definition of the indexing function allow the operation to be applied to the empty sequence?

Hint Answer

Membership

The **membership** operation, written $v \in s$ or v in s , checks whether value v is an element of sequence s . Here's one way to define it.

Function: membership

Inputs: $values$, a sequence; $value$, an object

Preconditions: true

Output: $is\ member$, a Boolean

Postconditions: $is\ member$ if and only if there's an integer $index$ such that $values[index] = value$

The postcondition states that the output is true only when there's an integer for which the indexing operation is defined and returns the input value. Note that the postcondition does *not* state how such an index can be found. In the previous chapters, the postconditions were similar to the algorithm that implemented the operation. But postconditions aren't algorithms: they are conditions – Boolean expressions that must be true after the algorithm does its job.

Postconditions *check* the output: they don't *compute* it.

We assume the membership operation has best-case complexity $\Theta(1)$ and worst-case complexity $\Theta(|values|)$. The reasoning is as follows. To decide whether $value$ is an element of $values$, the operation has to go through each member of $values$ and check if it's equal to $value$. The best-case scenario, when the operation does the least work, is when the first member of the sequence is $value$: the search is over after one comparison.

There are two worst-case scenarios, when the operation does the most work: the $value$ is the last item of the sequence or it doesn't occur at all. In both scenarios, the operation compares $value$ against all sequence members, i.e. it takes linear time in the length of the sequence: if the number of items doubles, the operation does double the work.

Comparison

If the items of a sequence are **pairwise comparable**, i.e. each item can be compared to every other item, then we can apply the minimum and maximum operations to determine the smallest and largest values.

The **lexicographic comparison** of two sequences does a pairwise comparison of items, one by one, until a decision can be made, i.e. items are compared until they differ or one sequence ends before the other. If two sequences are equal until one ends, then the shorter sequence is considered ‘less than’ the longer one. Some examples:

- $(1, 2) < (1, 2, 3)$ because the left sequence ends before the right one
- $(1, 3, 3) > (1, 2, 3)$ because the second item of the left sequence is greater than the second item of the right one
- $() < s$ for any non-empty sequence s
- $(1, 2, 3) \neq (1, \text{true})$ because the second items differ.

The last two sequences can only be compared for equality and inequality, because the other comparison operations aren’t defined for 2 and true.

As usual, we write $s1 \leq s2$ to mean $s1 < s2$ or $s1 = s2$, and similarly for other operations.

The comparison operations on sequences have best-case complexity $\Theta(1)$ because in the best-case scenario the first item of both sequences differ and a decision can be made after one comparison, which takes constant time. The comparisons have worst-case complexity $\Theta(\min(|left|, |right|))$ because in a worst-case scenario all items of the shorter sequence (or all except the last one) are equal to the corresponding items of the longer sequence and the decision is delayed until the end of the shorter sequence.

4.1.2 Creating sequences

The following operations create new sequences from existing ones.

Slicing

The **slicing** operation extracts a consecutive sequence of items from the input sequence s . The resulting **slice** is defined by the given start and end indices. We follow Python’s notation $s[start:end]$ with the understanding that the item at index end is *not* included. A slice is also called a **substring**. The term ‘subsequence’ has another meaning, explained in [Section 4.6](#).

For example, if $s = (6, 7, 8, 9)$, then $s[1:4]$ is the substring $(7, 8, 9)$, with the items from indices 1 to 3, inclusive. In other words, it’s the slice from the second to the fourth item. If s has fewer than four items, then the slice isn’t defined. Here’s a definition:

Function: slicing

Inputs: $values$, a sequence; $start$, an integer; end , an integer

Preconditions: $0 \leq start \leq end \leq |values|$

Output: $substring$, a sequence

Postconditions: if $start = end$, then $substring = ()$, otherwise $substring = (values[start], values[start + 1], \dots, values[end - 1])$

The preconditions allow the end index to be equal to the length of the sequence, so that the last item can be included in the slice. If $start = end$, then the slice is empty.

There are two reasons why the item at index end isn't included in the slice. First, it makes it easier to see how many items are in the slice. If the end item were included, the length of the slice would be $end - start + 1$. By not including it, $|s[start:end]| = end - start$, e.g. $s[2:7]$ has five items.

The second reason is to make it easier to split sequences. If you want to split a sequence s at index i , then the 'left' part of the sequence is $s[0:i]$ and the 'right' part is $s[i:|s|]$.

The slicing operation copies all items in the slice to a new sequence, so the complexity is linear in the size of the slice: $\Theta(end - start)$.

Concatenating

The **concatenation** operation, written $left + right$ in M269, forms a new sequence by joining both input sequences.

Function: concatenation

Inputs: $left$, a sequence; $right$, a sequence

Preconditions: true

Output: $joined$, a sequence

Postconditions: $joined = (left[0], \dots, left[|left| - 1], right[0], \dots, right[|right| - 1])$

If sequence s is empty, then there are no items $s[0], \dots, s[|s| - 1]$. For example, if $left$ is empty, then $joined = (right[0], \dots, right[|right| - 1])$. And if $right$ is also empty, then $joined = ()$.

The concatenation $left + right$ copies all items of $left$ and all items of $right$ to the new sequence, so the run-time is proportional to the number of items copied: $|left| + |right|$. The concatenation operation is linear in the total length of the inputs: $\Theta(|left| + |right|)$. If both inputs double their length, then the total number of items doubles, and so does the run-time of concatenation.

A sequence $pattern$ is a **prefix** of a sequence s if there's a sequence $rest$ such that $pattern + rest = s$. In other words, $pattern$ is a substring of s that starts at index 0 of s . Vice versa, if there's a sequence $rest$ such that $rest + pattern = s$, i.e. s ends with $pattern$, then $pattern$ is a **suffix** of s . For every sequence s , $() + s = s + () = s$. This means that the empty sequence $()$ is a prefix and a suffix (and thus a substring) of every sequence, and every sequence is a prefix and a suffix (and thus a substring) of itself.

Sorting

A sequence can be **sorted** in ascending (smallest to largest item) or descending order if the items are pairwise comparable. For example, $(3, 3, 7, 9)$ is in ascending order and $(9, 7, 3, 3)$ is in descending order. Here's one definition:

Function: ascending sort

Inputs: $original$, a sequence

Preconditions: all items in $original$ are pairwise comparable

Output: $sorted$, a sequence

Postconditions:

- *sorted* is a permutation of *original*
- $\text{sorted}[\text{index}] \leq \text{sorted}[\text{index}+1]$ for $\text{index} = 0, 1, \dots, |\text{sorted}| - 2$

A **permutation** is a rearrangement. The first postcondition relates the output to the input: the output has the same items as the input, possibly in a different order. (The input sequence may already be in ascending order.) The second postcondition defines what ascending order means.

In algorithms in English we write ‘s in ascending order’ or ‘s in descending order’ for some sequence s. The sorting functions produce a new sequence that can be part of a longer expression or be the right-hand side of an assignment, like

```
let sorted be sequence in ascending order
```

Sorting can be the basis for solving other problems, e.g. finding the median (the middle value when values are sorted) or the ten largest values. We’ll consider the complexity of sorting later.

4.2 Strings

Many applications process textual data or present text messages to the user. Text is formed from characters that represent letters, digits, punctuation, mathematical symbols, etc. A sequence of zero or more characters is a **string**, e.g. (H, i, !).

The usual, and more compact, notation is to write strings within single or double quotation marks, also known simply as quotes, e.g. ‘Hi!’ or “Hi!”. The quotes signal the start and end of the string. They’re not part of the string itself, in the same way that the parentheses and commas in (H, i, !) are not items of the sequence.

In M269, the string ADT is a restricted form of the sequence ADT, where the values are immutable strings. Python’s `str` type implements the string ADT.

4.2.1 Literals

Python follows the same notation as English. For example, we write ‘Hi!’ or “Hi!”. The double quote is the single character ”, not two single-quotes. Two single-quotes ‘ ’ (or two double quotes “”) denote the empty string, a sequence of no characters, a characterless string.

If a string includes a single-quote or apostrophe (they’re the same character), then it must be enclosed in double quotes, e.g. “It’s OK”. Vice versa, strings that include double quotes must be enclosed in single-quotes, e.g. ‘Kane whispered “Rosebud” as he died.’ If a string has both kinds of quotes, or spans multiple lines like docstrings, enclose it in three single or double quotes, e.g. '''Rhett said "Frankly, my dear, I don't give a damn." and left.'''

A string literal represents a value, hence it’s an expression, and therefore gets evaluated.

```
[1]: """Rick: And remember, this gun's pointed right at your heart.  
Louis: That is my least vulnerable spot."""
```

```
[1]: "Rick: And remember, this gun's pointed right at your heart.\nLouis:  
→That is my least vulnerable spot."
```

The interpreter shows an alternative literal for the multi-line string, with double quotes and newline characters (\n). Not very readable, I must say. The print function displays the string itself, without the enclosing quotes.

```
[2]: print("""Rick: And remember, this gun's pointed right at your heart.  
Louis: That is my least vulnerable spot.""")
```

```
Rick: And remember, this gun's pointed right at your heart.  
Louis: That is my least vulnerable spot.
```

String literals can have accented characters like ñ, é, ö and any character listed in the Unicode standard, which covers Chinese, Arabic and most other written languages.



Info: TM112 Block 1 Section 1.1.5 introduces the Unicode standard.

Mistakes

Typeset quotes are often curved, but in Python they're straight. If you copy curved quotes from a PDF, HTML or Word document into Python code, you get a syntax error.

```
[3]: 'Game over ...'  
  
Cell In[3], line 1  
      'Game over ...'  
      ^  
  
SyntaxError: invalid character ' ' (U+2018)
```

The starting quote isn't the ' character, so the interpreter doesn't recognise the start of a string literal, assuming instead it's an identifier with a strange character.

Starting a literal with one kind of quote and ending with another is also a syntax error.

```
[4]: 'holy guacamole!"  
  
Cell In[4], line 1  
      'holy guacamole!"  
      ^  
  
SyntaxError: unterminated string literal (detected at line 1)
```

Strings within single and double quotes can't span multiple lines.

```
[5]: "Rick: And remember, this gun's pointed right at your heart.  
Louis: That is my least vulnerable spot."  
  
Cell In[5], line 1  
      "Rick: And remember, this gun's pointed right at your heart.  
      ^
```

(continues on next page)

(continued from previous page)

```
SyntaxError: unterminated string literal (detected at line 1)
```

In both examples the interpreter complains that it reached the end of the line (EOL) before it reached the end of the string.

Jupyter notebooks and other code editors use syntax colouring to distinguish strings (in red), keywords (in bold green), operators (in bold purple), etc. That's useful to detect errors before running the code. When the whole string or part of it isn't in red, as in two of the examples, you know there's some mistake.

4.2.2 Inspecting strings

Python's `str` type supports many operations, in particular the length, indexing, comparison and membership operations.

Length

The function `len` returns the size of a string.

```
[6]: len('')      # length of the empty string
[6]: 0

[7]: len("""Hello!
""")
[7]: 8
```

Indexing

Python is more flexible than the indexing operation I defined: it allows any integer index.

```
[8]: 'hello'[0]      # retrieve the first character
[8]: 'h'

[9]: 'hello'[5 - 1]    # the index can be an integer expression
[9]: 'o'

[10]: 'hello'[-1]
[10]: 'o'
```

Negative indices count from the end of the string: the character at index `-1` is the first character from the end, the character at index `-2` is the second character from the end, etc. In M269, we shall use mainly index `-1`, because it's a convenient shorthand for `len(s) - 1` to access the last item of sequence `s`.

```
[11]: text = 'hello'
text[len(text) - 1]      # the same as text[-1]
[11]: 'o'
```

Comparisons

All six comparison operators apply to strings.

```
[12]: 'Tweedledee' == 'Tweedledum'
[12]: False

[13]: 'hello' < 'high'          # e comes before i, so 'hello' < 'high'
[13]: True

[14]: 'underpass' > 'under'
[14]: True
```

Python's lexicographic comparison uses the character ordering of the Unicode standard, which leads to results you may not expect.

```
[15]: 'aardvark' < 'Zeus'      # A-Z comes before a-z in Unicode
[15]: False

[16]: 'exposé' < 'exposed'    # accented letters come after non-accented
[16]: False
```

As long as you compare two strings *left* and *right* that only have the unaccented letters a to z and follow the same lower/uppercase pattern, then *left* < *right* is true if and only if dictionaries list *left* before *right*.

```
[17]: 'Aardvark' < 'Zeus'      # capital first letter, rest lowercase
[17]: True
```

The 26 lowercase letters are listed consecutively, in alphabetical order, in the Unicode standard. We can thus use comparisons to check if a character is a lowercase letter.

```
[18]: character = '!'
'a' <= character <= 'z'      # is the character a lowercase letter?
[18]: False
```

We can write similar expressions to check if a character is a digit or an uppercase letter.

We can apply the minimum and maximum operations to obtain the character that appears first or last in the Unicode table.

```
[19]: min('Zeus')          # in Unicode, A-Z comes before a-z
[19]: 'Z'
```

```
[20]: max('By Jove!')      # in Unicode, space and ! come before A-Z
[20]: 'y'
```

Membership

Python's `in` operator checks if the left operand is a substring of the right operand.

```
[21]: 'pose' in 'exposed'
[21]: True
```

```
[22]: 'hello' in 'Hello, world!'      # h and H are different characters
[22]: False
```

When the left operand is a single character, this becomes the membership operation.

```
[23]: ',' in 'Hello, world!'    # does the string contain a comma?
[23]: True
```

A Boolean expression of the form `not (substring in string)` can also be written as `substring not in string`, which is easier to read.

```
[24]: 'hello' not in 'Hello, world!'
[24]: True
```

Exercise 4.2.1

Assume `character` is a string of length 1. Write a Boolean expression that is true if and only if `character` is a decimal digit. Don't use comparisons.

```
[25]: character = '6'
# replace this by your expression
```

After running your code (you should get `True`), change the character to a letter and rerun the code (you should get `False`).

Hint Answer

Mistakes

A frequent mistake is to forget that indices start from zero. This leads to 'off by one' errors where you intend to access the n -th item of a sequence but are instead accessing the next item. If the index is 'out of bounds', the interpreter raises an **index error**, but if it's not, the 'off by one' error may only become apparent far later in the execution of the algorithm.

```
[26]: 'hello'[5] # there's no 6th character, counting from the start
-----
→-----
IndexError                                         Traceback (most recent)
←call last)
Cell In[26], line 1
----> 1 'hello'[5] # there's no 6th character, counting from the
←start

IndexError: string index out of range
```

```
[27]: 'hello'[-6] # there's no 6th character, counting from the end
-----
→-----
IndexError                                         Traceback (most recent)
←call last)
Cell In[27], line 1
----> 1 'hello'[-6] # there's no 6th character, counting from the end

IndexError: string index out of range
```

If we apply an operation to operands of the wrong type, we get a **type error**.

```
[28]: 'five' < 5
-----
→-----
TypeError                                         Traceback (most recent)
←call last)
Cell In[28], line 1
----> 1 'five' < 5

TypeError: '<' not supported between instances of 'str' and 'int'
```

Exercise 4.2.2

Explain whether these expressions are valid or lead to a syntax, type or index error:

1. "hello"['e']
2. ''[0]
3. len('goodman)[-1])
4. 42 in 'Everyone wears jersey 42 on Jackie Robinson Day'

Hint Answer

4.2.3 Creating strings

Python's `str` type also supports the concatenation and slicing operations, again with some 'extensions'.

Concatenation

Python overloads the addition operator to also represent concatenation:

```
[29]: 'Hello,' + 'world!'      # concatenation doesn't add spaces  
[29]: 'Hello,world!'
```

```
[30]: 'Hello' + ',', ' + 'world' + '!'  
[30]: 'Hello, world!'
```

Multiplication is just repeated additions, e.g. $3 \times 4 = 4 + 4 + 4$. By analogy, Python reuses the multiplication operator for repeated concatenation.

```
[31]: 3 * 'Ho'      # thus spoke Father Christmas  
[31]: 'HoHoHo'
```

```
[32]: 'Ho' * 0      # repeating zero times produces the empty string  
[32]: ''
```

We'll use repeated concatenation mainly for creating long test strings.

Exercise 4.2.3

What are the best- and worst-case scenarios of repeated concatenation? What are the corresponding complexities?

Hint Answer

Slicing

Python allows arbitrary integers as the start and end indices. The character at the end index is excluded from the slice.

```
[33]: 'hello'[0:1]      # indices 0 to 0, i.e. 'hello'[0]  
[33]: 'h'  
  
[34]: 'hello'[1:4]      # indices 1 to 3, i.e. 2nd to 4th character  
[34]: 'ell'  
  
[35]: 'hello'[3:3]      # if start = end, the slice is empty
```

```
[35]: ''
[36]: 'hello'[2:1]          # if start > end, the slice is empty
[36]: ''
[37]: 'hello'[2:-1]         # 3rd to penultimate character (-1 not included)
[37]: 'll'
```

The following examples show why the last index isn't included.

```
[38]: 'hello'[1:4]          # len(text[start:end]) = end - start
[38]: 'ell'
[39]: 'hello'[0:1] + 'hello'[1:4]      # text[a:b] + text[b:c] = text[a:c]
[39]: 'hell'
```

As an example of slicing, here's how to swap the two halves of a string.

```
[40]: text = 'Alice and Bob'
       middle = len(text) // 2
       text[middle:len(text)] + text[0:middle]
[40]: 'and BobAlice '
```

In Python, the start index of a slice can be omitted (it defaults to zero) and so can the end index (it defaults to the length of the string). In other words, `s[:i]` is the same as `s[0:i]` and `s[i:]` is the same as `s[i:len(s)]`. The previous expression to swap both halves can be more succinctly written as:

```
[41]: text[middle:] + text[:middle]
[41]: 'and BobAlice '
```

I think this shows more clearly that the result is the text from the middle onwards, followed by the text up to the middle.

Exercise 4.2.4

Explain why the swapping code works for strings of length 0 and 1, the edge cases, without raising an index error.

Hint Answer

Conversion

Each Python data type has a **constructor**, a function with the same name as the type to create a value of that type. Constructor `str` creates a string from a number:

```
[42]: str(2020)
```

```
[42]: '2020'
```

Exercise 4.2.5

What is the complexity of converting an integer to a string?

Hint Answer

Mistakes

The `+` operator means addition if both operands are numbers, and concatenation if both are strings. The `*` operator means multiplication if both operands are numbers, and repeated concatenation if one is an integer and the other is a string. Any other combination leads to a type error. The exact message may vary.

```
[43]: 3 + '3'
```

```
-----  
TypeError
```

```
    ↪call last)
```

```
Cell In[43], line 1
```

```
----> 1 3 + '3'
```

```
Traceback (most recent)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
[44]: 'high' * 5.0
```

```
-----  
TypeError
```

```
    ↪call last)
```

```
Cell In[44], line 1
```

```
----> 1 'high' * 5.0
```

```
Traceback (most recent)
```

```
TypeError: can't multiply sequence by non-int of type 'float'
```

You may think repeated concatenation doesn't make sense for negative integers, but the Python designers decided to produce the empty string instead of raising an error.

```
[45]: 'Ho' * -5
```

```
[45]: ''
```

Exercise 4.2.6

Assume that you have a string variable `text` and an integer variable `times` with a positive value. Write an expression that repeats the text the given number of times, separated by ellipses (...).

```
[46]: text = 'hello'  
times = 3  
# replace this by your expression
```

Running the previous cell should create the string 'hello...hello...hello'. If you run the cell with `times = 1`, you should obtain just 'hello'.

Hint Answer

4.3 Iteration

There are two main ways of repeating instructions: a for-loop executes the instructions a fixed number of times; a while-loop repeatedly executes the instructions while a condition is true. Python supports both kinds of loops. I'll cover them through very simple examples that focus on how to express iteration in English and in Python. Full problems are left for a later section.



Info: TM112 introduces iteration in Block 1: for-loops in Section 2.2 and while-loops in Section 4.5.

4.3.1 For-loops

Here's an algorithm that displays some text vertically, one character per line:

1. let `text` be 'hello'
2. for each `character` in `text`:
 1. print `character`

This is translated to Python by just dropping the word 'each':

```
[1]: text = "hello"  
for character in text:  
    print(character)
```

```
h  
e  
l  
l  
o
```

Python's for-loop can only iterate over sequences. So, if we want to iterate over some numbers, we must create an integer sequence in Python. Python's `range` data type represents a sequence of consecutive integers. The corresponding constructor `range(start, end)` creates the

sequence `start, start+1, ..., end-1`. The constructor behaves like the slicing operation: the start number is included, but the end number isn't. This means that the sequence is empty if `start >= end`.

Here's a simple algorithm that prints some even numbers:

1. for each *number* from 1 to 5:
 1. print $2 \times \text{number}$

This is translated to Python as follows:

```
[2]: for number in range(1, 6): # the end number is set one higher
      print(2 * number)

2
4
6
8
10
```

Some further examples:

```
[3]: for number in range(-2, -1):
      print(number)

-2
```

```
[4]: for number in range(3, 3): # no output if start = end
      print(number)
```

```
[5]: for number in range(0, -1): # no output if start > end
      print(number)
```

```
[6]: for number in range(4): # start = 0 if omitted
      print(number)

0
1
2
3
```

Since the upper bound is excluded, and the lower bound is zero by default, `for index in range(len(s))` iterates over the indices of string `s`.

```
[7]: text = "hello"
for index in range(len(text)):
    print(text[index])

h
e
l
```

(continues on next page)

(continued from previous page)

```
1  
o
```

The corresponding English algorithm is:

1. let $text$ be ‘hello’
2. for each $index$ from 0 to $|text| - 1$:
 1. print $text[index]$

It’s possible to iterate backwards. Here’s a countdown algorithm:

1. for each $number$ from 10 down to 1:
 1. print $number$

This can be translated to Python like so:

```
[8]: for number in range(10, 0, -1):  
    print(number)  
  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1
```

The third argument of `range` is the stepwise increment or decrement between consecutive numbers in the sequence, e.g. `range(start, end)` is shorthand for `range(start, end, 1)`. The end number is always excluded from the sequence, so here we must say the end is zero. Without the `-1`, the loop wouldn’t execute, as the start (10) is larger than the end (0).

Exercise 4.3.1

Change step 2 of this algorithm so that the string is printed backwards, character by character.

1. let $text$ be ‘hello’
2. for each $index$ from 0 to $|text| - 1$:
 1. print $text[index]$

Translate step 2 of your algorithm to Python.

```
[9]: text = "hello"  
for index in range(): # put the correct arguments in  
    print(text[index])
```

```
-->-----  
TypeError  
    <--call last)  
Cell In[9], line 2  
    1 text = "hello"  
----> 2 for index in range(): # put the correct arguments in  
        3     print(text[index])  
  
TypeError: range expected at least 1 argument, got 0
```

Hint Answer

4.3.2 While-loops

A for-loop executes a fixed number of times, while a while-loop executes as long as some condition is true. Strictly speaking, for-loops are unnecessary because something like

1. for *number* from *start* to *end*:
 1. do something with *number*

can be rewritten as:

1. let *number* be *start*
2. while *number* \leq *end*:
 1. do something with *number*
 2. let *number* be *number* + 1

However, it's best to use for-loops rather than while-loops when the number of iterations is fixed, as for-loops make that clearer.

Consider the problem of writing all the positive even numbers until some limit *highest*, given as input. We don't know in advance how many numbers to print, so a while-loop is the natural choice.

1. let *number* be 2
2. while *number* \leq *highest*:
 1. print *number*
 2. let *number* be *number* + 2

This translates directly to Python:

```
[10]: highest = 6 # or some other value  
number = 2  
while number <= highest:  
    print(number)  
    number = number + 2
```

```
2
4
6
```

For this particular problem, it's possible to use a for-loop in Python, making use of the stepwise increment of the `range` constructor.

```
[11]: highest = 6
for number in range(2, highest + 1, 2): # 2 to highest in steps of 2
    print(number)
```

```
2
4
6
```

In general, for ‘open-ended’ iterations we must use a while-loop. For example, consider printing the outstanding mortgage, until it’s paid off. Let’s assume the inputs are the annual rate (a percentage), the monthly fixed payment, and the starting mortgage. Every month we must add the interest and deduct the payment, like so:

1. while $mortgage \neq 0$:
 1. let $interest$ be $mortgage \times rate / 12$
 2. let $mortgage$ be $mortgage + interest - payment$
 3. print $mortgage$

Contrary to for-loops, while-loops may never stop. If the mortgage never becomes exactly zero, which is likely, we have an **infinite loop**. For this problem, to avoid looping forever we must guarantee that the mortgage decreases, i.e. have the precondition $payment > mortgage \times rate / 12$, and change the while condition to $mortgage > 0$.



Info: The mortgage problem and infinite loops are introduced in TM112 Block 1 Section 4.5.

4.3.3 Repeat-loops

Sometimes we want to repeat some instructions a fixed number of times. In those cases we use a ‘repeat n times’ loop, like this one:

1. repeat 3 times:
 1. print ‘echo...’

Python doesn’t have this kind of loop, so it must be translated to a for-loop:

```
for times in range(3):
    print('echo...')
```

It takes a moment to realise that the variable isn't used within the loop. The English description makes this much clearer by not having a variable at all.

A repeat-until loop iterates at least once, until a condition becomes true. There's always at least one mortgage payment, so we can use a repeat-until loop.

1. repeat:
 1. let *interest* be $mortgage \times rate / 12$
 2. let *mortgage* be *mortgage* + *interest* - *payment*
 3. print *mortgage*
2. until $mortgage \leq 0$

The Boolean expression in a while-loop is a continuation condition (if it's true, the loop carries on), whereas the Boolean expression in a repeat-until loop is a stopping condition (if it's true, the loop stops). Every repeat-until loop can be written as a while-loop:

1. repeat:
 1. do something
2. until *condition*

is equivalent to

1. let *stop* be false
2. while not *stop*:
 1. do something
 2. let *stop* be *condition*

4.3.4 Nested loops

Any instructions can be executed repeatedly, including other loops. Writing the times table up to some input number n requires nested loops.



Info: TM112 introduces nested loops and a times table program in Block 1 Section 2.5.

1. for each *left* from 1 to n :
 1. for each *right* from 1 to n :
 1. let *product* be *left* × *right*
 2. print *left* ‘×’ *right* ‘=’ *product*

Step 1.1 is executed for each value of *left*, so the successive products are 1×1 , 1×2 , ..., $1 \times n$, 2×1 , ..., $2 \times n$, 3×1 , etc. Step 1.1.2 prints each line of the table, e.g. ' $2 \times 3 = 6$ '.

Steps 1.1.1 and 1.1.2 take constant time and are executed $n \times n$ times. The overall run-time is proportional to n^2 . We say the algorithm has **quadratic complexity**: $\Theta(n^2)$. More generally, if a

loop does i iterations and each iteration has complexity $\Theta(e)$, then the loop has complexity $\Theta(i \times e)$. In this example, the inner loop has complexity $n \times \Theta(1) = \Theta(n)$ and the outer loop has complexity $n \times \Theta(n) = \Theta(n^2)$.



Note: The complexity of a loop is the number of iterations multiplied by the complexity of the body of the loop.

Here's the corresponding Python code:

```
[12]: n = 3 # or some other positive integer
for left in range(1, n + 1):
    for right in range(1, n + 1):
        product = left * right
        print(left, "x", right, "=", product)

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

Like `min` and `max`, `print` can take one or more arguments. It prints them separated by spaces. Arguments can be any integer, Boolean or string expressions, so the last argument can be the multiplication expression. Moreover, when there are no arguments, `print` produces an empty line. Putting all this together, here's a more pleasing program and result.

```
[13]: n = 3 # or some other positive integer
for left in range(1, n + 1):
    for right in range(1, n + 1):
        print(left, "x", right, "=", left * right)
    print()

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6

3 x 1 = 3
3 x 2 = 6
```

(continues on next page)

(continued from previous page)

```
3 x 3 = 9
```

The indentation makes the first `print` part of the inner loop and the second one part of the outer loop.

It's possible to have more than two nested loops and to have while-loops within for-loops and vice versa.

4.4 Linear search

Many problems using sequences are **search problems**: they involve finding one or more elements of the sequence that satisfy some condition. Such problems can be solved with a **linear search**, an algorithm that goes systematically through the sequence and checks each element. This section shows some examples of linear searches at work and *en passant* illustrates some finer points of the problem-solving process.



Info: TM112 Block 2 Section 2.3 introduces search problems and algorithmic patterns for them.

4.4.1 Finding characters

Imagine that we have a non-empty string with one or more sentences, each ending with a full stop. We're asked to create a new string with just the first sentence.

Although this isn't formulated as a search problem, it involves a search: finding the first full stop in the string. Once we know its index, we simply take the slice up to that index: that's the first sentence.



Note: Even if a problem isn't stated as a search problem, think whether doing a search could solve it.

Problem definition and instances

Let's take the opportunity to define the more general problem of finding the first occurrence of a given character in a given string.

Function: first index

Inputs: `text`, a string; `character`, a string

Preconditions: $|character| = 1$

Output: `index`, an integer

Postconditions: if $character$ in $text$, $index$ is the smallest integer such that $text[index] = character$, otherwise $index = |text|$

Search problems often have postconditions of the form ‘... is the first / last / smallest / largest ... such that ...’ or similar. To indicate that the character doesn’t occur in the text, the output is set to an impossible index.

For the test table, I need to think of edge cases. An input sequence with the smallest allowed length is always an edge case. I also need to create tests for different options of where and how often $character$ occurs in $text$.

Case	$text$	$character$	$index$
smallest input	“	‘a’	0
occurs at start	‘all’	‘a’	0
occurs in the middle	‘abracadabra’	‘c’	4
occurs at the end	‘hi!’	‘!’	2
multiple occurrences	‘abracadabra’	‘b’	1
no occurrence	‘abracadabra’	‘k’	11

Algorithm

The linear search algorithm simply goes through all the indices of the text and stops when it finds the character. The output is the index at which it stopped. Otherwise, the postconditions tell us the output is the length of the string.

1. for each $index$ from 0 to $|text| - 1$:
 1. if $text[index] = character$:
 1. stop
 2. let $index$ be $|text|$

Complexity

Whenever there’s a stop statement within a loop, we must think of best- and worst-case scenarios: under which conditions does the algorithm stop the earliest and stop the latest?

For this algorithm, the loop can stop in its first iteration if the first character of the text is the sought character. In that case step 1.1, which takes constant time, is executed once, so the algorithm has best-case complexity $\Theta(1)$.

In a worst-case scenario the algorithm goes through all characters. This may happen because the character doesn’t occur at all or only in the last position. Step 1.1 is executed $|text|$ times; the worst-case complexity is $\Theta(|text|)$.

The complexity of linear search algorithms is often, but not always, constant in the best case and linear in the size of the sequence in the worst case.

Code and tests

The translation to Python makes use of the `range` constructor. We must not forget that the end of the range isn't included and so must be one higher (or lower, if iterating backwards) than we need.

```
[1]: def first_index(text: str, character: str) -> int:  
    """Return the lowest index of character in text.  
  
    Preconditions: len(character) = 1  
    Postconditions: if text includes character, then the output is  
    the lowest index such that text[index] = character,  
    otherwise the output is len(text)  
    """  
  
    for index in range(len(text)):  
        if text[index] == character:  
            return index  
    return len(text)
```

```
[2]: first_index("", "a") == 0
```

```
[2]: True
```

```
[3]: first_index("all", "a") == 0
```

```
[3]: True
```

```
[4]: first_index("abracadabra", "c") == 4
```

```
[4]: True
```

```
[5]: first_index("hi!", "!") == 2
```

```
[5]: True
```

```
[6]: first_index("abracadabra", "b") == 1
```

```
[6]: True
```

```
[7]: first_index("abracadabra", "k") == 11
```

```
[7]: True
```

Performance

To illustrate the use of repeated concatenation, let's check that the worst-case complexity is linear in the size of the input string. A worst-case scenario is for the sought character to not occur.

```
[8]: text = 100 * "blah" # start with a not too short string
%timeit -r 3 -n 1000 first_index(text, '!')
text = 200 * "blah"
%timeit -r 3 -n 1000 first_index(text, '!')
text = 400 * "blah"
%timeit -r 3 -n 1000 first_index(text, '!')
text = 800 * "blah"
%timeit -r 3 -n 1000 first_index(text, '!')

12.4 µs ± 1.45 µs per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
20.5 µs ± 836 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
42.5 µs ± 406 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
87.5 µs ± 169 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
 ↴each)
```

On my computer, the run-time roughly doubles as the input size doubles, thereby confirming that the worst-case complexity is linear.

4.4.2 Valid password

Consider the problem of deciding whether a given string is a valid password, which we take to mean having at least one lowercase letter and one digit. To keep the example short, I focus on the problem definition, algorithm and complexity only.

Function: valid password

Inputs: *password*, a string

Preconditions: true

Output: *is valid*, a Boolean

Postconditions: *is valid* if and only if *password* contains a digit and a lowercase letter

There are two conditions to be satisfied, so we should write at least four tests, with inputs that satisfy both, neither or just one of the conditions.

Exercise 4.4.1

Write a test table. Add rows as necessary.

Case	password	is valid

Hint Answer

Algorithm

This is a decision problem that can be solved by searching for two characters with certain properties. We can use linear search again, as long as we remember if we found a lowercase letter and a digit so far. There are only two states for each, found or not found, so Boolean variables will do. This problem doesn't require keeping track of indices, so we can iterate over the string directly.

1. let *has letter* be false
2. let *has digit* be false
3. for each *character* in *password*:
 1. if '`0`' \leq *character* \leq '`9`':
 1. let *has digit* be true
 2. if '`a`' \leq *character* \leq '`z`':
 1. let *has letter* be true
4. let *is valid* be *has digit* and *has letter*

This is a typical use of Boolean variables as **flags**. A flag is 'raised', i.e. the Boolean is set to true, when some condition occurs and it stays raised to remember that the condition occurred. The use of Boolean flags is common in searches.

Exercise 4.4.2

Explain whether the following algorithm is correct.

1. let *has letter* be false
2. let *has digit* be false
3. for each *character* in *password*:
 1. let *has digit* be '`0`' \leq *character* \leq '`9`'
 2. let *has letter* be '`a`' \leq *character* \leq '`z`'
4. let *is valid* be *has digit* and *has letter*

Hint Answer

Exercise 4.4.3

The algorithm goes through the whole string even if a lowercase letter and digit appear early on in the string. Alice and Bob are modifying the algorithm to stop as soon as it can. This is Alice's algorithm:

1. let *has letter* be false
2. let *has digit* be false
3. for each *character* in *password*:
 1. if '`0`' \leq *character* \leq '`9`':

1. let *has digit* be true
2. if ‘a’ ≤ *character* ≤ ‘z’:
 1. let *has letter* be true
 3. if *has digit* and *has letter*:
 1. stop
4. let *is valid* be *has digit* and *has letter*

This is Bob’s algorithm:

1. let *has letter* be false
2. let *has digit* be false
3. for each *character* in *password*:
 1. if ‘0’ ≤ *character* ≤ ‘9’:
 1. let *has digit* be true
 2. if ‘a’ ≤ *character* ≤ ‘z’:
 1. let *has letter* be true
3. let *is valid* be *has digit* and *has letter*
4. if *is valid*:
 1. stop

For each algorithm, explain whether it’s correct or not.

Answer

Complexity

The complexity of an algorithm is an indication of how its run-time grows for increasingly large inputs. By definition, a constant-time step doesn’t make the run-time grow and so doesn’t contribute to the complexity of the algorithm. We can thus ignore all constant-time steps when analysing the complexity. Well, not quite all: we can’t ignore if and stop statements, because they affect how the algorithm behaves. For the original algorithm, and Alice’s, we ignore steps 1, 2, 3.1.1, 3.2.1 and 4. (Bob’s algorithm doesn’t have a step 4: we ignore step 3.3 instead.)

The ‘partial’ algorithm

3. for each *character* in *password*:
 1. if ‘0’ ≤ *character* ≤ ‘9’:
 2. if ‘a’ ≤ *character* ≤ ‘z’:

has exactly the same complexity as the complete algorithm. Both if-statements take constant time, as each does one or two comparisons. Whether the current character is a letter, digit or something else, each iteration takes constant time. The complexity is thus linear in the number of iterations, which is the length of the input string: $\Theta(|\text{password}|)$.

As mentioned before, the complexity of an algorithm is not about the run-times for particular problem instances, but rather about the growth of the run-times for instances with increasingly large values or sizes. Therefore, a scenario is a *collection* of problem instances with increasing sizes or values: a scenario is not a *single* problem instance. Even though this algorithm does the least work for the empty string, because the loop is skipped, the empty string is *not* a best-case scenario.

Exercise 4.4.4

What are best- and worst-case scenarios for a linear search algorithm that stops as soon as it knows the password is valid?

Hint Answer

Exercise 4.4.5

Implement and test the password-validation function. You can choose the original algorithm or the more efficient version that stops early.

```
[9]: # replace this with your code
```

Add code cells for the tests.

Answer

4.5 Tuples and tables

In Python, a tuple is an immutable sequence of items, which can be of any type, including other tuples. This allows for nested tuples, which we can use to represent tabular data.

4.5.1 Literals and operations

Tuples are written within parentheses, with items separated by commas. For example () is the empty tuple and (1, (2, 3), True) is a tuple with three items: an integer, a tuple with two integers and a Boolean.

Python's `tuple` data type supports the same operations as `str`, with the same notation, so a few brief examples suffice.

```
[1]: len((1, (2, 3), True))
```

```
[1]: 3
```

```
[2]: (1, (2, 3), True)[1] # indexing: 2nd item is a pair of integers
```

```
[2]: (2, 3)
```

```
[3]: (1, 2, 3) < (1, 4, 3) # comparison: item by item
```

```
[3]: True
```

```
[4]: (1, (2, 3), True)[1:3] # slicing: 2nd to 3rd items
[4]: ((2, 3), True)
```

```
[5]: min((3, 4, -2, 7, 9)) # extra parentheses for the tuple
[5]: -2
```

For tuples, the `in` operator checks for membership, not for a substring like with strings.

```
[6]: 2 in (1, 2, 3) # (1, 2, 3) does have member 2
[6]: True
```

```
[7]: (1, 2) in (1, 2, 3) # (1, 2, 3) doesn't have member (1, 2)
[7]: False
```

```
[8]: (1, 2) in ((1, 2), 3) # ((1, 2), 3) does have member (1, 2)
[8]: True
```

The `tuple` constructor creates a tuple from another sequence.

```
[9]: tuple("hello")
[9]: ('h', 'e', 'l', 'l', 'o')
```

```
[10]: tuple(range(1, 4))
[10]: (1, 2, 3)
```

As tuples are sequences, we can directly iterate over them:

```
[11]: for item in (1, (2, 3), True):
        print(item)

1
(2, 3)
True
```

4.5.2 Mistakes

Forgetting a comma between items, or forgetting an item between two commas, usually leads to a syntax error. However, consider this example:

```
[12]: (1(2, 3), True)
<>:1: SyntaxWarning: 'int' object is not callable; perhaps you
     ↴missed a comma?
<>:1: SyntaxWarning: 'int' object is not callable; perhaps you
                                         (continues on next page)
```

(continued from previous page)

```
→missed a comma?  
/var/folders/b2/13mhcbxn2b3fsdkb4d9lh32jc84fk1/T/ipykernel_71728/  
→3144750586.py:1: SyntaxWarning: 'int' object is not callable;  
→perhaps you missed a comma?  
(1(2, 3), True)
```

```
-----  
→----  
TypeError Traceback (most recent  
→call last)  
Cell In[12], line 1  
----> 1 (1(2, 3), True)  
  
TypeError: 'int' object is not callable
```

This strange type error is due to the missing comma after the first item. The interpreter thinks that 1 is meant to be a function with arguments (2, 3) and duly reports that we can't call an integer, only functions.

The error messages of most programming language interpreters are sometimes baffling. They can misguide us in finding the error's source. I've been making errors on purpose to show various 'diseases' (errors) and their corresponding 'symptoms' (messages). I recommend you do the same.



Note: When learning a programming language, make deliberate mistakes in your code to associate error messages with their causes.

A single item within parentheses is a normal expression with redundant parentheses, not a tuple with a single item. To let the interpreter know we mean a tuple, we must add a comma.

```
[13]: (3) + (4) # arithmetic expression with redundant parentheses
```

```
[13]: 7
```

```
[14]: (3,) + (4,) # concatenation of tuples of length 1
```

```
[14]: (3, 4)
```

Applying the `tuple` constructor to a Boolean or number results in a type error, because it can only be applied to sequences.

```
[15]: tuple(True)
```

```
-----  
→----  
TypeError Traceback (most recent  
continues on next page)
```

(continued from previous page)

```

→call last)
Cell In[15], line 1
----> 1 tuple(True)

TypeError: 'bool' object is not iterable

```

The error message states that Boolean values can't be iterated over and so it's not possible to create a tuple.

4.5.3 Tables

Data is often arranged in tabular form. We can use tuples to represent tables. A table is a sequence of rows, each row being a sequence of cells. Here's a small selection of board games I enjoy, with one game per row.

```
[16]: games_by_row = ( # each item is a row
    ("Board game", "Rating", "Owned"), # header row
    ("Power Grid", 10, True), # first data row
    ("Vintage", 8, True),
    ("Pandemic", 9, False),
)
```

The header row is a tuple of strings; the other rows are tuples of strings, integers and Booleans.

I formatted the tuples like tables, with right-aligned values, to better show the structure of the nested tuples. But you don't have to: the interpreter doesn't care about spacing around tuple items.

```
[17]: games_by_row
[17]: (('Board game', 'Rating', 'Owned'),
      ('Power Grid', 10, True),
      ('Vintage', 8, True),
      ('Pandemic', 9, False))
```

To access a single cell's value, we must first index the row and then the column within that row. Let's suppose we want to check whether I own Power Grid.

```
[18]: row_index = 1 # Power Grid is in the 2nd row
column_index = 2 # ownership is in the 3rd column
row = games_by_row[row_index]
cell = row[column_index]
cell
[18]: True
```

The first indexing operation selects a row of the table; the second indexing operation selects a cell of that row. The following line is equivalent to the above step-by-step process.

```
[19]: games_by_row[1][2] # 2nd row, 3rd column  
[19]: True
```

Indexing is left-associative: each operation takes the result produced by the previous operation. The next example shows that the order of indices matters.

```
[20]: games_by_row[2][1] # 3rd row (Vintage), 2nd column (Rating)  
[20]: 8
```

The first index always selects a row; the second index always selects a column.

The header row is at index zero, so each game is at its ‘natural’ index: the first game is at index 1, the second game is at index 2, etc. However, it’s easy to get confused about which index corresponds to which column for wide tables. Giving them memorable names helps.

```
[21]: GAME = 0  
RATING = 1  
OWNED = 2
```

```
[22]: games_by_row[1][OWNED]  
[22]: True
```

```
[23]: games_by_row[2][RATING]  
[23]: 8
```

I can hear you exclaim ‘Variable names should be in lowercase!’ The convention in Python is that if a variable name is all in uppercase, then the value should not be further changed: the variable should be treated like a **constant**. Other programming languages have proper constants, i.e. a second assignment to the constant raises an error, but Python has constants by convention.

Exercise 4.5.1

We can organise the data differently, with one game per column.

```
[24]: games_by_column = ( # each item is a column  
    ("Board game", "Power Grid", "Vintage", "Pandemic"),  
    ("Rating", 10, 8, 9),  
    ("Owned", True, True, False),  
)
```

Write an expression that indexes this table to retrieve my rating for Pandemic. Use constants. The expression should evaluate to 9.

```
[25]: # replace with your expression
```

Hint Answer

4.5.4 Iterating

Tabular data is processed with iterative algorithms that go through one or more rows or columns, depending on the problem. For example, the following algorithm computes the mean rating of the games.



Info: MU123 Unit 4 Section 3.2 and TM112 Block 2 Section 2.4.2 introduce the mean and how to compute it.

```
[26]: total = 0
count = 0
for row in range(1, len(games_by_row)):    # skip header row
    count = count + 1
    total = total + games_by_row[row][RATING]
print(total / count)

9.0
```

This algorithm works for any number of games in the table, as long as there's at least one, to avoid division by zero.

A slightly simpler and faster algorithm computes the number of games instead of counting them one by one.

```
[27]: total = 0
for row in range(1, len(games_by_row)):    # skip header row
    total = total + games_by_row[row][RATING]
print(total / (len(games_by_row) - 1))

9.0
```

To go through all cells of a table, we need nested loops: one to iterate over the rows and the other to iterate over the columns.

```
[28]: for row in games_by_row:
        for cell in row:
            print(cell)
```

Board game

Rating

Owned

Power Grid

10

True

Vintage

8

True

Pandemic

(continues on next page)

(continued from previous page)

```
9  
False
```

To iterate first by column and then by row requires indexing:

```
[29]: for column in range(len(games_by_row[0])):  
    for row in range(len(games_by_row)):  
        print(games_by_row[row][column])
```

```
Board game  
Power Grid  
Vintage  
Pandemic  
Rating  
10  
8  
9  
Owned  
True  
True  
False
```



Info: TM351 introduces more sophisticated data types to represent and analyse tabular data.

Exercise 4.5.2

The next tuple represents the state of a Noughts and Crosses game after three turns of each player. (You don't need to know the game to solve this exercise.)

```
[30]: tic_tac_toe = (  
    ('X', 'O', 'X'),  
    (' ', 'X', ' '),  
    ('O', ' ', 'O')  
)
```

The number of empty spaces (character ' ') can be used to determine whose turn it is next or whether the game has ended. Write an algorithm in Python that displays the number of empty spaces for a board represented in the variable `tic_tac_toe`. You should obtain the value 3 for the board above. Change the tuple and rerun both cells to test with another board configuration.

```
[31]: # replace with your code
```

Hint Answer

Exercise 4.5.3

This exercise is about how to make use of the data types you learned so far to represent some data in a way that eases the implementation of some operations.

Moksha Patam, better known as Snakes and Ladders, is an ancient Indian board game, played usually on a 10×10 board, with positions numbered from 1 to 100, and some snakes and ladders connecting pairs of positions.

A player moves their pawn forward by rolling a die. If the pawn lands on the bottom of a ladder, it immediately moves forward to the position at the top of the ladder. If it lands on the head of a snake, it immediately moves backward to the position at the tail of the snake. Then it's the next player's turn. The first player to reach the position 100 (which has no snake head) wins.

How would you represent a board and its configuration of snakes and ladders with tuples? Do you have to represent the board as a table, with nested tuples? Assuming that the current position of each pawn is in a separate variable, how would the position be represented?

Devise a representation for the board and pawn positions that makes it easy to implement the movement of pawns.

Hint Answer

4.6 Lists

Python provides various sequence types. The ones we've seen so far (`str`, `range` and `tuple`) are immutable: the sequence can't be modified. Python's `list` data type is the most flexible one: **lists** are mutable and may have items of different types. All operations on tuples work in the same way on lists, so I won't repeat them. This section focuses on the operations that change lists, so let's first revisit the sequence ADT.



Info: Some texts use 'list' as a synonym of 'sequence'. In M269 the term refers to Python's data type.

4.6.1 Modifying sequences

Besides operations to inspect and create sequences, we need operations to remove and add individual items. They modify the input sequence, i.e. they aren't mathematical functions. To define them, we replace 'Function:' with 'Operation:' in the template and add an entry 'Inputs/Outputs:' for variables that have their value changed by the operation. When we must distinguish the value of an input/output variable x before and after the operation, we use $\text{pre-}x$ and $\text{post-}x$ respectively. Here's how we can define the operation to remove the item at a given index i , which we write as 'remove s_i ' or 'remove $s[i]$ ' in algorithms in English.

Operation: remove

Inputs/Outputs: $values$, a sequence

Inputs: $index$, an integer

Preconditions: $0 \leq \text{index} < |\text{values}|$

Postconditions: $\text{post-values} = (\text{pre-values}[0], \dots, \text{pre-values}[\text{index} - 1], \text{pre-values}[\text{index} + 1], \dots, \text{pre-values}[|\text{pre-values}| - 1])$

The postcondition states that the value at the given index has ‘disappeared’ but all other items remain in the same order. Without the ‘pre-’ and ‘post-’ indications, the postcondition would be ambiguous. For example, if the sequence has length 5 before an item is removed, it has length 4 afterwards, so which length would $|\text{values}|$ refer to? Writing $|\text{pre-values}|$ or $|\text{post-values}|$ makes it clear.

We assume that removing the item at index i from a sequence of length n is implemented by copying each of the subsequent $n - i - 1$ items one position down. Also, the length has to be updated whenever the sequence changes, so that the size operation can just look it up in constant time. Copying values in RAM and subtracting one from an integer take constant time, so the removal operation does $n - i - 1 + 1$ constant-time operations. The complexity is $\Theta(n - i)$ or $\Theta(|\text{values}| - \text{index})$.

Using the pre- x notation in preconditions or complexity expressions is unnecessary. By definition, they can only refer to input values.

A **subsequence** is obtained by deleting zero or more items from the input sequence. Every substring is a subsequence, but not every subsequence is a substring. For example, $(1, 3, 5)$ is a subsequence of $(1, 2, 3, 4, 5)$ but not a substring, because 1, 3 and 5 aren’t consecutive items in the longer sequence.

The operation to add an item, more precisely to insert it at a given position, can be defined like this:

Operation: insert

Inputs/Outputs: values , a sequence

Inputs: index , an integer; value , an object

Preconditions: $0 \leq \text{index} \leq |\text{values}|$

Postconditions: $\text{post-values} = (\text{pre-values}[0], \dots, \text{pre-values}[\text{index} - 1], \text{value}, \text{pre-values}[\text{index}], \dots, \text{pre-values}[|\text{pre-values}| - 1])$

The postcondition defines what inserting at a certain position means: to shift all items from that position onwards to the next position, in order to ‘make space’ for the new item. As the postcondition explicitly shows, $\text{post-values}[\text{index}] = \text{value}$. If $\text{index} = |\text{pre-values}|$, as the preconditions allow, then the item is effectively added to the end of the sequence. This special case of insertion is so common it has a name: **appending**. In algorithms in English, we write these operations respectively as

- insert value in values at index
- append value to values .

The insertion operation shifts not just the items after the index, but also the item at the index itself, so it copies $n - i$ values and increments the length. The complexity is therefore $\Theta(|\text{values}| - \text{index} + 1)$. In the grand scheme of things, i.e. for long sequences, the operation spends most of its time shifting items. The update of the length hardly impacts the overall run-time. So, we

write simply $\Theta(|values| - index)$. More generally, a fixed number doesn't affect the growth of the run-time.



Note: In complexity analysis, $\Theta(e + c) = \Theta(e - c) = \Theta(e)$ if c is an integer constant and e is an expression involving the input values or sizes.

If the value is appended then the complexity is $\Theta(1)$, as no shifting takes place.

Replacing the item at a given index with a new item can be achieved by first removing it and then inserting the new item at the same index. Shifting all subsequent items first down and then up is very inefficient. The assignment 'let s_i be x ' does the replacement in constant time.

Exercise 4.6.1

The following defines a function that outputs the reverse of the input sequence.

Function: reversed sequence

Inputs: $values$, a sequence

Preconditions: true

Output: $reversed$, a sequence

Postconditions: $reversed = (values[|values| - 1], values[|values| - 2], \dots, values[1], values[0])$

Modify the definition so that it reverses the input sequence instead of creating a new sequence.

Hint Answer

4.6.2 Creating lists

Let's now see how Python implements the sequence ADT with type `list`.

Like with tuples and strings, lists can be created in various ways. The simplest is to write a list literal. It looks like a tuple literal but with square brackets instead of parentheses. The empty list is `[]` and `[1, [2, 3], ('A', 'B'), True]` is a list with four elements of different types: an integer, a list of integers, a tuple of strings and a Boolean. Contrary to tuples, a list of length 1 doesn't need an extra comma, because the square brackets can't be confused with redundant parentheses.

As with strings, printing a list and displaying it (by just writing the variable name) produces different outputs.

```
[1]: to_do = [
    'finish writing this chapter',
    'write the next one',
    'rinse and repeat for another 20+ chapters'
]
to_do
```

```
[1]: ['finish writing this chapter',
      'write the next one',
      'rinse and repeat for another 20+ chapters']

[2]: print(to_do)

['finish writing this chapter', 'write the next one', 'rinse and
repeat for another 20+ chapters']
```

Lists can also be created by slicing or concatenating existing lists. Repeated concatenation is particularly convenient for creating long lists with all elements initialised to the same value.

```
[3]: 10 * [0]  # create an integer list initialised to zeros

[3]: [0, 0, 0, 0, 0, 0, 0, 0, 0]
```

We can use the `list` constructor to convert another sequence to a list.

```
[4]: list(range(1, 10))

[4]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

[5]: list("Hello, world!")

[5]: ['H', 'e', 'l', 'l', 'o', ' ', ' ', 'w', 'o', 'r', 'l', 'd', '!']

[6]: board = (("X", "O", "X"), (" ", " ", " "), ("O", "X", " "))
list(board) # doesn't convert nested tuples

[6]: [('X', 'O', 'X'), (' ', ' ', ' '), ('O', 'X', ' ')]
```

Finally, we can use the `sorted` function to obtain a sorted list from any sequence of pairwise comparable items. The function has a second optional Boolean parameter to indicate whether to sort in descending order.

```
[7]: sorted("Hello, world!")

[7]: [' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'o', 'o', 'r', 'w']

[8]: sorted([2, 4, -3, 4.1])

[8]: [-3, 2, 4, 4.1]

[9]: sorted([2, 4, -3, 4.1], reverse=True) # sort in descending order

[9]: [4.1, 4, 2, -3]
```

We'll look at sorting algorithms in [Chapter 14](#). For the moment we assume that sorting takes linear time in the length of the sequence in the best case and quadratic time in the worst case. In the best case, a single pass over the sequence detects that it's already sorted. In the worst case, a sorting algorithm must compare every item to every other item, which takes a quadratic number of comparisons, to know where each item appears in the sorted sequence.

4.6.3 Mistakes

In Python, some arguments of some functions have to be named: you can't just pass a value. If you do, the interpreter doesn't know what to do with it and says that the function is being called with too many arguments.

```
[10]: sorted([1, 2, 3], True) # sort in descending order
-----
↔-----  

TypeError Traceback (most recent)
  ↵call last)
Cell In[10], line 1
----> 1 sorted([1, 2, 3], True) # sort in descending order

TypeError: sorted expected 1 argument, got 2
```

Forgetting a comma before a nested list makes the interpreter think we're trying to index the previous item because indexing uses square brackets too. This leads to a type error if the item's type doesn't include the indexing operation.

```
[11]: [1[2, 3], True] # comma missing after 1
<>:1: SyntaxWarning: 'int' object is not subscriptable; perhaps you
  ↵missed a comma?
<>:1: SyntaxWarning: 'int' object is not subscriptable; perhaps you
  ↵missed a comma?
/var/folders/b2/13mhcbxn2b3fsdkb4d9lh32jc84fk1/T/ipykernel_71734/
  ↵3918117735.py:1: SyntaxWarning: 'int' object is not subscriptable;-
  ↵perhaps you missed a comma?
  [1[2, 3], True] # comma missing after 1
-----
↔-----  

TypeError Traceback (most recent)
  ↵call last)
Cell In[11], line 1
----> 1 [1[2, 3], True] # comma missing after 1

TypeError: 'int' object is not subscriptable
```

You can't concatenate sequences of different types.

```
[12]: ["first task", "second task"] + "third task"
-----
↔-----  

TypeError Traceback (most recent)
  ↵call last)
Cell In[12], line 1
----> 1 ["first task", "second task"] + "third task"
```

(continues on next page)

(continued from previous page)

TypeError: can only concatenate list (not "str") to list

```
[13]: (1, 2) + [3, 4]
```

```
-----  
TypeError  
  ↗call last)  
Cell In[13], line 1  
----> 1 (1, 2) + [3, 4]
```

Traceback (most recent)

TypeError: can only concatenate tuple (not "list") to tuple

4.6.4 Modifying lists

A list can be changed by replacing, removing, or inserting a new item. As mentioned above, we replace an item with an assignment, e.g.

1. let *daily temperature* be $[-5, -2, 0, 1, -1]$
2. let *daily temperature*[1] be -4

In Python this becomes

```
[14]: daily_temperature = [-5, -2, 0, 1, -1]  
       daily_temperature[1] = -4
```

The list has indeed changed:

```
[15]: daily_temperature  
[15]: [-5, -4, 0, 1, -1]
```

The `insert` method adds an item at a given position. A **method** is a function that is only known in the context of a particular data type. Whereas `print` and `len` are functions that can be applied to various data types, `insert` only applies to lists, so the syntax is different. It uses **dot notation**: first write an expression (typically a variable) of the required data type, then a dot, then the method name with the remaining inputs in parentheses. In the case of the `insert` method, the remaining inputs are first the index and then the item to be inserted.

```
[16]: daily_temperature.insert(0, -6) # insert -6 at index 0  
       daily_temperature  
[16]: [-6, -5, -4, 0, 1, -1]
```

```
[17]: daily_temperature.insert(-1, "week 2:")  
       daily_temperature
```

```
[17]: [-6, -5, -4, 0, 1, 'week 2:', -1]
```

Hmm, this didn't work. I wanted to add the text to the end of the list, but instead it got into the penultimate position. Can you explain why and figure a way of making it appear last?

The operation worked as described. I asked to insert the text in the last position, so the value at that position shifted right. Let's start afresh.

```
[18]: daily_temperature = [-5, -2, 0, 1, -1]
```

The insertion operation takes the index of where the item will be put. If I want it to appear after the current last item then I can't use the index of that item: I must use the next index.

```
[19]: daily_temperature.insert(len(daily_temperature), "week 2:")
daily_temperature
```

```
[19]: [-5, -2, 0, 1, -1, 'week 2:']
```

The `append` method adds an item to the end of a list: `a_list.append(an_item)` is short for `a_list.insert(len(a_list), an_item)`.

```
[20]: daily_temperature.append(6)
daily_temperature
```

```
[20]: [-5, -2, 0, 1, -1, 'week 2:', 6]
```

To remove an item, we use the `pop` method, indicating the index of the item to be removed. For convenience, the method returns the item that was removed.

```
[21]: daily_temperature.pop(0) # remove first item
```

```
[21]: -5
```

```
[22]: daily_temperature.pop(-1) # remove last item
```

```
[22]: 6
```

```
[23]: daily_temperature
```

```
[23]: [-2, 0, 1, -1, 'week 2:']
```

If you want to see a method's documentation, you must use dot notation to indicate which data type has that method.

```
[24]: help(list.insert)
```

```
Help on method_descriptor:
```

```
insert(self, index, object, /) unbound builtins.list method
    Insert object before index.
```

Lists have a further method to sort a list, instead of creating a new sorted list. It has the same optional parameter as `sorted`.

```
[25]: numbers = [1, 4, -2, 3]
numbers.sort()
numbers
```

```
[25]: [-2, 1, 3, 4]
```

```
[26]: numbers.sort(reverse=True)
numbers
```

```
[26]: [4, 3, 1, -2]
```

Exercise 4.6.2

We can also append an item using concatenation:

```
[27]: daily_temperature = daily_temperature + [6]
```

From a complexity point of view, is this a good idea?

Hint Answer

Exercise 4.6.3

Here's the board games table again, with one game per row, but this time as a list of lists, so that it can be modified.

```
[28]: games_by_row = [
    ['Board game', 'Rating', 'Owned'],
    ['Power Grid', 10, True],
    ['Vintage', 8, True],
    ['Pandemic', 9, False]
]
```

Write code that adds one more column with game prices and one more row with another game. Use fictitious data. The new column should be the right-most column. The new game should be listed first, before Power Grid.

```
[29]: # replace this by your code to change the table
games_by_row # display the new table
```

Hint Answer

4.6.5 Mistakes

Replacing, removing or adding an item of an immutable type leads to a type error: the operation is attempted on a type that doesn't support it.

```
[30]: text = "i love grilled fish"
text[0] = "I"

-----
↔-----  

TypeError                                         Traceback (most recent)
↑call last)
Cell In[30], line 2
    1 text = "i love grilled fish"
----> 2 text[0] = "I"  

TypeError: 'str' object does not support item assignment
```

Calling a method without using dot notation leads to a name error, because there's no such function in the 'global' context.

```
[31]: pop([1, 2, 3], 1) # there's no pop function ...

-----
↔-----  

NameError                                         Traceback (most recent)
↑call last)
Cell In[31], line 1
----> 1 pop([1, 2, 3], 1) # there's no pop function ...

NameError: name 'pop' is not defined
```

A method's name is only known in the context of a particular data type. The interpreter must know the type before it's told the method name. The dot notation does exactly that: the interpreter, which reads code left to right like you and I do, first looks at the type of the expression before the dot and then checks whether that data type has a method with the name after the dot.

```
[32]: [1, 2, 3].pop(1) # but lists do have a method named pop
[32]: 2
```

4.7 Reversal

This section presents a single problem, reversing a sequence, to further illustrate sequences, iteration and the problem-solving process.

4.7.1 Problem definition

This is the definition from the *previous section*.

Function: reverse sequence

Inputs: *values*, a sequence

Preconditions: true

Output: *reverse*, a sequence

Postconditions: *reverse* = (*values*[| *values* | - 1], *values*[| *values* | - 2], ..., *values*[1], *values*[0])

With the Python operations we have seen so far, it's impossible to write a Python function that reverses any sequence, be it a string, tuple or list. We have to restrict the problem to a particular data type. I'll solve it for lists and I'll leave strings to you as an exercise. (Tuples are handled similarly.)

Function: reverse list

Inputs: *values*, a list

Preconditions: true

Output: *reverse*, a list

Postconditions: *reverse* = [*values*[| *values* | - 1], *values*[| *values* | - 2], ..., *values*[1], *values*[0]]

Since this function is for a Python data type, I use Python's notation (square brackets for lists) in the postcondition.

4.7.2 Problem instances

I have to think of some problem instances to test the function. The smallest possible inputs are always edge cases and must be included in the test table. For this example, it's the empty list. If the preconditions allow the empty sequence, then a sequence with a single item is an edge case too: it's the smallest non-empty sequence.

For problems about sequences, it's often convenient to test sequences of odd and even length, because the middle element of a list of odd length may be treated differently. In this problem, the middle member is the only one that has the same position in the reverse list.

Test cases for sequences should also include, if the preconditions allow, duplicate and unique items, and values of different types.

When thinking about problem instances, put your hacker hat on: you're trying to break the algorithm to reveal it's incorrect. Throw curveballs: think of valid inputs that most people wouldn't dream of when reading the problem description. You don't need large inputs to properly test an algorithm. An algorithm is often incorrect because it failed to consider a particular case, e.g. all items in a sequence being the same. Such cases can be covered with small inputs. When it comes to problem instances for testing, think small, think wildly. (But not too wildly: all test cases must satisfy the preconditions.)

So far, we wrote test tables in Markdown and translated them to one code cell per test case. We can now write them directly in Python, as a list (or tuple) of test cases, each represented by a list or tuple. I prefer to write test tables as a list of tuples, so that I can later append a test case if I forgot one, but you can use any combination you prefer.

The table's name is the operation's name followed by `_tests`. Each row is the test case description (a string), followed by the input values and ending with the expected output value. The column headings are a comment instead of a row; you'll see why when we get to the actual testing.

Here's a possible table for the reversal problem. It includes odd- and even-length lists, values of different types, and lists with duplicate items.

```
[1]: reverse_list_tests = [
    # case,           values,           reverse
    ('empty list',   [],             []),
    ('length 1',     [4],            [4]),
    ('length 2',     [5, True],      [True, 5]),
    ('length 5',     [5, 6, 7, 8, 9], [9, 8, 7, 6, 5]),
    ('same items',   [0, 0, 0],      [0, 0, 0])
]
```

The `algoesup` module, part of the M269 software, has a function `check_tests` that checks if the test table is well formed. It takes two arguments: the table and a list with the types of the inputs and output. In this example, there's only one input (a list to be reversed) and the output is also a list (the reversed one), so we write:

```
[2]: import algoesup

algoesup.check_tests(reverse_list_tests, [list, list])
OK: the test table passed the automatic checks.
```

When we need a function `f` from a module `m`, we can write `from m import f` to directly refer to `f`, instead of writing `m.f`.

```
[3]: from algoesup import check_tests

check_tests(reverse_list_tests, [list, list])
OK: the test table passed the automatic checks.
```

Function `check_tests` only spots structural mistakes, like the table not being a list or tuple of tests, or a test not starting with a string describing the test case. Even if the table is OK, the tests may be wrong. For example, the checks won't spot if the expected output is wrong.

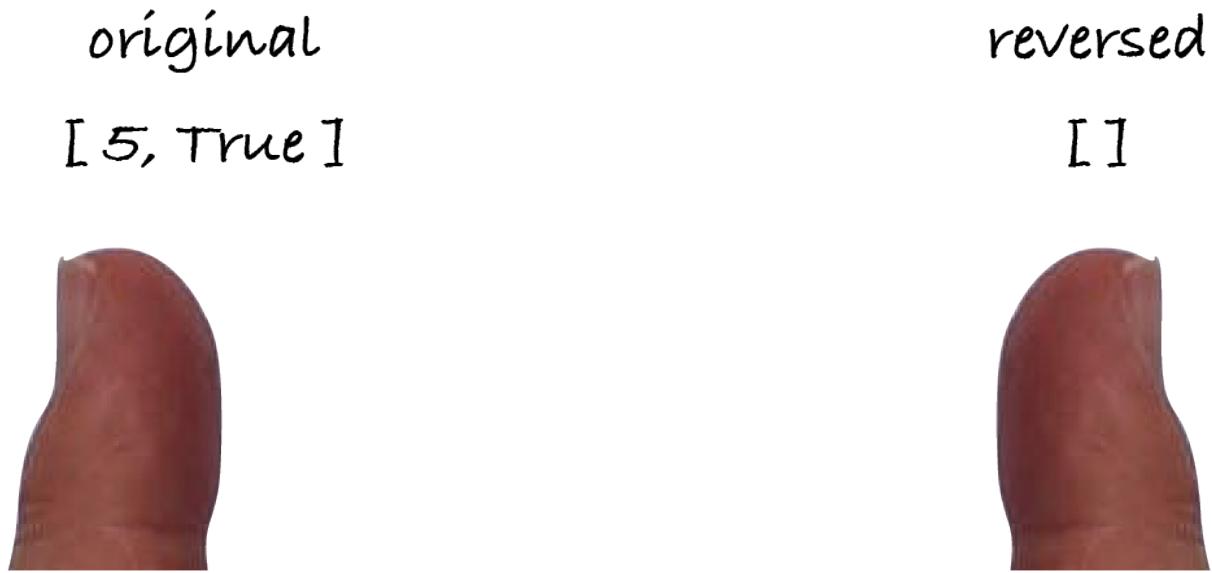
4.7.3 Algorithm

Sometimes the best way to come up with an algorithm is to think how we'd do it manually. And I literally mean with our hands.

The reverse operation takes one list and produces another one. Lists have to be processed item by item. I use my left index finger to point at the item being processed in the input list and my right index finger to point to the position where that item should be put in the output list.

Initially, my left finger points at the first item of *values* and my right finger points to an empty *reverse* list. Let's use the length 2 test case.

Figure 4.7.1



The first two steps are obvious: insert the item pointed by the left finger into the empty list and move the left finger to the next item.

Figure 4.7.2



The second item of *values* should be inserted at the start of *reverse*, hence I can keep the right finger where it is.

Figure 4.7.3



If the input list were longer, I would continue in the same way. Each item of *values* has to be inserted at index 0 of *reverse* to push the previous items to the right. I'm ready to write the algorithm:

1. let *reverse* be the empty list
2. for each *item* in *values*:
 1. insert *item* at index 0 of *reverse*

Before implementing this algorithm, let's check it works for the edge cases. Does it work for lists of length 0 and 1?

Yes, it does. The loop is executed as often as the length of the input list, so the output is the same as the input for lists of length 0 and 1.

4.7.4 Complexity

I can ignore step 1 because it takes constant time. Step 2 is executed $|values|$ times. The complexity of inserting an item at index i in a list of length l is $\Theta(l - i)$. In step 3, $i = 0$, so that step has complexity $\Theta(|reverse|)$: it shifts all items in *reverse* up to make space for a new item at index 0. The complexity of the loop is hence $|values| \times \Theta(|reverse|) = \Theta(|values| \times |reverse|)$. Unfortunately, I can't write it like that because *reverse* isn't an input. Fortunately, $|reverse| = |values|$ because reversing a list doesn't change its length. The algorithm has quadratic complexity: $\Theta(|values|^2)$.

4.7.5 Code

The translation of the function definition and the algorithm to Python is:

```
[4]: def reverse_list(values: list) -> list:
    """Return the same items as values, in inverse order.
```

(continues on next page)

(continued from previous page)

```
Postconditions: the output is
[values[-1], values [-2], ..., values[1], values[0]]
"""
reverse = []
for item in values:
    reverse.insert(0, item)
return reverse
```

4.7.6 Tests

Having put the test cases in a table, we can automatically run all tests, instead of manually writing one code cell for each. This is done by function `algoesup.test`, which takes the function to be tested and the table of tests.

```
[5]: from algoesup import test

test(reverse_list, reverse_list_tests)
Testing reverse_list...
Tests finished: 5 passed (100%), 0 failed.
```

The `test` function goes through each row of the test table, extracts the case description, the input and the expected output, calls the `reverse_list` function on that input and compares the result to the expected output. If the actual and expected outputs differ, the test fails and is reported.

The `test` function checks the test table before running the tests, but does fewer checks than `check_tests`, namely it doesn't check the types of the tests' inputs and output. Despite this, I will use mostly function `test` to check the table, and only call `check_tests` when `test` can't be used, namely when I've yet to write the function to be tested.



Info: The `algoesup` module was written by former M269 student Michael Snowden and myself, to support the writing of algorithmic essays in Jupyter notebooks. If you have the time, I recommend you write an essay on a problem of your choice, to practice M269 concepts and your communication skills. See our [website](#) for example essays, essay templates, and guidance to get you started.

4.7.7 Performance

An algorithm with quadratic complexity takes much longer than an algorithm with linear complexity. Let's assume an algorithm with complexity $\Theta(e)$ does exactly e operations, each taking one microsecond. Here are the run-times for various input sizes n .

n	$\Theta(1)$	$\Theta(n)$	$\Theta(n^2)$
10	1 μ s	10 μ s	100 μ s
1,000	1 μ s	1 ms	1 s
2,000	1 μ s	2 ms	4 s
2,000,000	1 μ s	2 s	4,000,000 s = 46 days

When the input size doubles, a linear algorithm takes double the time, but a quadratic algorithm takes $2^2 = 4$ times as long. If the input is a thousand times as long then a linear algorithm takes a thousand times as long, but a quadratic algorithm takes $1000^2 = 1,000,000$ times as long!



Note: When the input size doubles, the run-time of algorithms with constant, linear and quadratic complexity respectively stays the same, doubles or quadruples.

To measure the run-times of quadratic algorithms we can't use very large inputs, unless we're prepared to wait quite a bit.

```
[6]: size = 10
for measurement in range(10):
    numbers = list(range(size)) # list [0, 1, 2, ..., size-1]
    print("Reversing", size, "numbers:")
    %timeit -r 5 reverse_list(numbers)
    size = size * 2

Reversing 10 numbers:
264 ns ± 1.09 ns per loop (mean ± std. dev. of 5 runs, 1,000,000 loops each)
Reversing 20 numbers:
499 ns ± 3.2 ns per loop (mean ± std. dev. of 5 runs, 1,000,000 loops each)
Reversing 40 numbers:
1.13 µs ± 6.59 ns per loop (mean ± std. dev. of 5 runs, 1,000,000 loops each)
Reversing 80 numbers:
3.33 µs ± 24.7 ns per loop (mean ± std. dev. of 5 runs, 100,000 loops each)
Reversing 160 numbers:
9.91 µs ± 28.4 ns per loop (mean ± std. dev. of 5 runs, 100,000 loops each)
Reversing 320 numbers:
31.9 µs ± 192 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
Reversing 640 numbers:
111 µs ± 309 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops each)
```

(continues on next page)

(continued from previous page)

```
Reversing 1280 numbers:  
409 µs ± 2.12 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops  
→each)  
Reversing 2560 numbers:  
1.58 ms ± 19.7 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops  
→each)  
Reversing 5120 numbers:  
6.18 ms ± 30.2 µs per loop (mean ± std. dev. of 5 runs, 100 loops  
→each)
```

On my machine the run-times start quadrupling for the larger values: the run-times for very small inputs are not a reliable indication of complexity. I should have started with a size of, say, 500, and do fewer than ten measurements.

Exercise 4.7.1

Write a more efficient algorithm to produce a reverse list. (The next exercise asks you to justify why it's more efficient.)

Hint Answer

Exercise 4.7.2

Analyse the complexity of your algorithm, showing that it's more efficient than the original algorithm.

Answer

Exercise 4.7.3

Translate your algorithm to Python and test it.

```
[7]: def reverse_list_2(values: list) -> list:  
    """Return the same items as values, in inverse order.  
  
    This is a more efficient version of reverse_list.  
    Postconditions: the output is  
    [values[-1], values [-2], ..., values[1], values[0]]  
    """  
    # replace with your function body  
  
test(reverse_list_2, reverse_list_tests)
```

Hint Answer

Exercise 4.7.4

Write a reversal algorithm for when *values* and *reverse* are strings.

Hint Answer

Exercise 4.7.5

What is the complexity of your reversal algorithm for strings?

Hint Answer

Exercise 4.7.6

Write an algorithm in English that reverses a list **in-place**, i.e. without creating a new list. There's a single input/output variable *values*. (See the solution to [Exercise 4.6.1](#).) Think with your hands.

Hint Answer

Exercise 4.7.7

Implement your algorithm in the next code cell and run it.

Note that the header indicates that the function returns `None`. That's Python's way of saying that it returns nothing, because the reversal is done in-place. Like `True` and `False`, `None` is both a value (that can be compared with the equality and inequality operations) and a keyword (so that it can't be used as a variable name by mistake). In Python, all functions that haven't a `return` statement return `None`.

```
[8]: def reverse_in_place(values: list) -> None:
    """Write the docstring."""
    # replace this with your code
    # modify `values` variable and do NOT use a return statement

    # `algoesup.test` can only test functions that return values,
    # so I've written the testing code for you
    for test in reverse_list_tests:
        name = test[0]
        values = test[1]
        reverse = test[2]
        reverse_in_place(values)
        if values != reverse:
            print(name, "FAILED:", values, "instead of", reverse)
    print("Tests finished.")
```

Answer

4.8 Optional practice

This section suggests further problems, in case you feel you need more practice. You may not have the time to attempt all these problems. In the wording of these and future problems, ‘implement’ implies documenting, testing and analysing the complexity too. You should practise measuring the run-times with loops that double the inputs for at least one problem. To save time, you may write the problem definition directly as a function header and docstring.

Like all exercises, you can discuss these freely with your tutor and in the forums, unless they are part of TMA questions. If you want others to comment on your solution, then include the word ‘solution’ in your post’s title so that your peers can avoid seeing solutions until they have attempted the problem themselves.

The problems are from easiest to hardest in my opinion, but your mileage may vary. I recommend you read the next section (summary) and the *next chapter* (advice for TMA questions) before attempting the problems here.

These and all other optional exercises have deliberately no answers in the book.

4.8.1 DNA

DNA is the molecule that contains an organism’s genetic code. It consists of two strands (entwined in a double helix) of the bases adenine, cytosine, guanine and thymine. Each strand can be modelled as a string with letters A, C, G and T.

Exercise 4.8.1

Implement a function that checks if a string models a DNA strand, i.e. if it only includes some or all of the four uppercase letters.

Hint

4.8.2 Minimum

If Python didn’t implement the minimum and maximum functions on sequences with pairwise comparable items, we could do so ourselves. Alice implements the minimum operation with a linear search for the smallest item. Bob first sorts the input sequence.

Exercise 4.8.2

Implement both approaches for one sequence type of your choice (string, tuple or list) and explain which one is more efficient.

Function names should state *what* the function does, not *how* it works. However, when implementing different algorithms for the same problem, you can name your Python functions to reflect the algorithm, e.g. `min_with_search` and `min_with_sorting`.

Hint

4.8.3 Lexicographic comparison

If Python only supported comparisons of numbers, Booleans and characters, we'd have to implement lexicographic comparison for sequences.

Exercise 4.8.3

Implement the equality comparison for strings. You can use the comparison operators on single characters.

Hint

4.8.4 Palindrome

A palindrome is a text that can be read backwards in the same way as forwards. The strings 'level' and 'step on no pets' are palindromes. We assume that the left-to-right and right-to-left texts have to match exactly: 'step on NO pets' isn't a palindrome under this definition.

Exercise 4.8.4

Implement a function that decides whether a string is a palindrome. Don't modify the input string.

Hint

4.8.5 Mode

The **mode** is the most frequent item in a sequence. For example, the mode of 'Hello there' is 'e'. If several items occur equally frequently then the sequence may have multiple modes, but let's assume the input sequence has a single mode.

Exercise 4.8.5

Implement a function that computes the mode of a string.

Hint

4.8.6 Images

Images can be represented as tables, and notebooks support multiple media, so with a little bit of Python you can generate and manipulate images.

An RGB raster image can be represented as a two-dimensional grid of pixels, each represented by a triplet of integers from 0 to 255 indicating the intensity of red, green and blue light. For example, a green pixel is represented by $(0, g, 0)$, with g from 1 to 255. If red, green and blue light have the same intensity, then the resulting colour is on the grey scale: $(0, 0, 0)$ represents black, $(127, 127, 127)$ a medium grey and $(255, 255, 255)$ represents white.

To help you manipulate images, I have written some auxiliary code. All auxiliary code files used in this book are named `m269_...py` and are in the same `notebooks` folder as the 'root' notebook `M269.ipynb`. You won't need to open any auxiliary file.



Note: Do **not** modify the auxiliary Python files `m269_...` as that may break the notebooks that use them.

The file `m269_image.py` has some functions and constants to support some simple image processing:

- `load_image(filename)` creates a grid from the given BMP, PNG or JPEG file
- `save_image(grid, filename)` saves the grid to a BMP, PNG or JPEG file
- `new_image(width, height, colour)` returns a monochromatic grid
- `show_image(grid)` displays the grid in the notebook
- `width(grid)` returns the number of columns (horizontal number of pixels)
- `height(grid)` returns the number of rows (vertical number of pixels).

A grid is a table (list of lists) of RGB triplets (tuples).



Info: The functions use the [Python Image Library](#) and [matplotlib](#) to load, display and save an image.

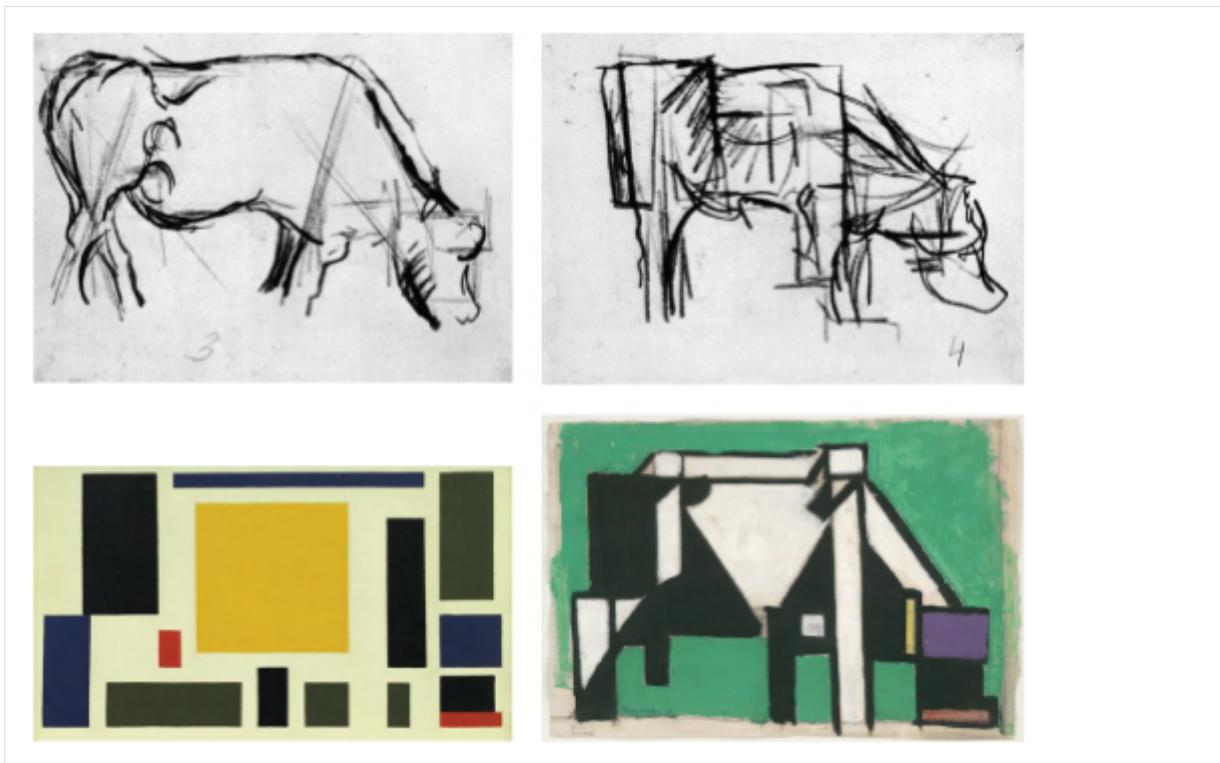
We must load the `notebooks/m269_image.py` file before we can use its functions. That's done with the IPython command `%run -i ../m269_test` in the first line of a code cell. The command executes the code in the given file as if the code were in the cell: it is similar to an import statement. The relative path `../` tells IPython that the file is in the parent folder of (i.e. the folder above) this notebook. IPython automatically adds the `.py` extension.

Here's how to load and display the M269 cover image.

```
[1]: %run -i ../m269_image

cover = load_image("../cover.jpg")
print("Image has", width(cover), "x", height(cover), "pixels.")
show_image(cover)

Image has 498 x 355 pixels.
```

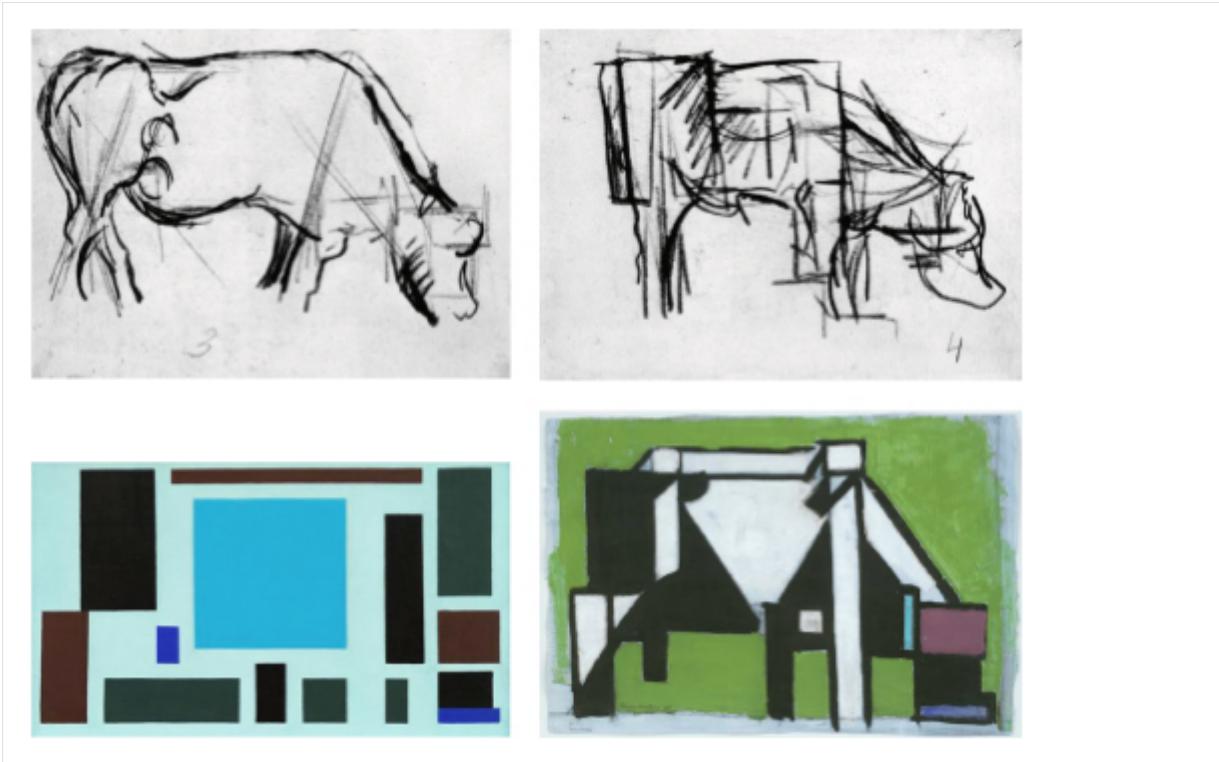


The grid (table) returned by `load_image` stores the pixels in row order. The column increases from left to right and the row increases from top to bottom. This means that `image[0][0]` (the top left RGB tuple of table `image`) represents the top left pixel of the image, and `image[h-1][w-1]` represents the bottom right pixel, where `h` and `w` are the number of rows (height) and columns (width), respectively.

The `R`, `G` and `B` constants have value 0, 1 and 2 respectively, so that you can easily access each colour of a pixel. Here's an example.

```
[2]: def swap_red_blue(image: list) -> None:
    """Swap red and blue components of each pixel."""
    for column in range(width(image)):
        for row in range(height(image)):
            pixel = image[row][column]
            image[row][column] = (pixel[B], pixel[G], pixel[R])

swap_red_blue(cover)
show_image(cover)
```



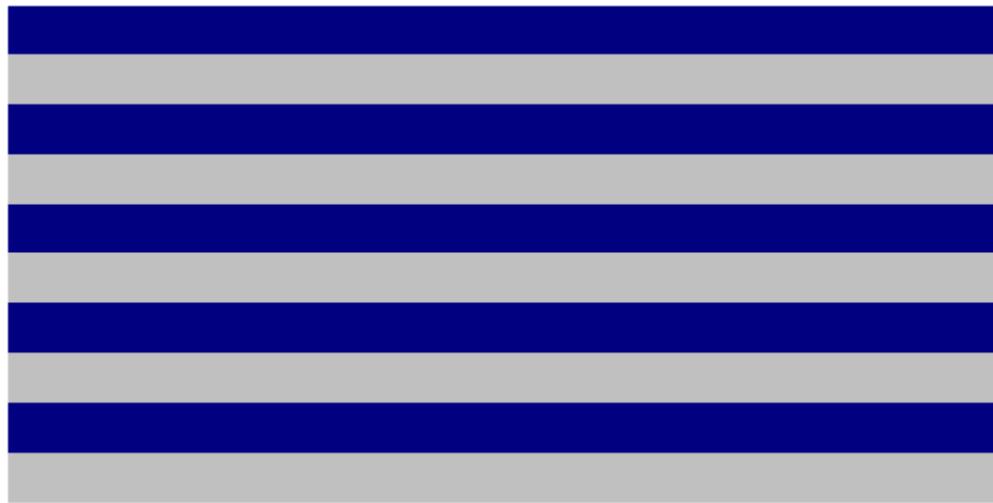
Colours that are a mixture of red and green, like yellow, became blueish, while those that are a mix of blue and green became brownish.

You can also create an image from scratch. I've defined constants for the 16 basic colours in [HTML 4](#). Here's an example.

```
[3]: def stripes(n: int, colour1: tuple, colour2: tuple) -> list:
    """Create n pairs of alternating stripes, each 100 x 5 pixels.

    Preconditions: n > 0; colour1 and colour2 are RGB triplets
    Postconditions: the top stripe has colour1
    """
    image = new_image(100, n * 2 * 5, colour2)
    # add 5 rows of colour1 every 10 rows
    for row in range(0, height(image), 10):
        for increment in range(0, 5):
            image[row + increment] = [colour1] * 100
    return image

image = stripes(5, NAVY, SILVER)
show_image(image)
```



The image is only 100×50 pixels, but it's shown larger than the cover image: `show_image` automatically adapts the display to large and small images.

Exercise 4.8.6

Implement this function to transform colour images to greyscale images:

Operation: colour to grey

Inputs/Outputs: *image*, an RGB raster image

Preconditions: true

Postconditions: if a pixel is (r, g, b) in pre-image, then it is $(grey, grey, grey)$ in post-image, with $grey = \text{floor}((r + g + b) / 3)$

Test the function by applying it to the M269 cover image.

```
[4] : # replace this with your code
```

4.9 Summary

A **sequence** is an ordered collection of zero or more **items**, also called the sequence's **members** or **elements**. The empty sequence has no members. A sequence is **sorted** if it is ordered by ascending or descending value. This requires all items to be **pairwise comparable**. If a sequence can't be modified, it's **immutable**; otherwise, it's **mutable**. If two sequences have the same items, but possibly in a different order, then each sequence is a **permutation** of the other. **Strings** are sequences of characters.

Python's `str`, `range`, `tuple` and `list` types implement immutable strings, immutable sequences of integers, immutable sequences and mutable sequences, respectively. Tuples and lists can contain items of any type, in particular other tuples or lists. This can be used to represent tables.

4.9.1 Sequence operations

The following tables list the operations supported by the sequence ADT and Python's data types, and their assumed complexities in M269. If two complexities are listed, they're the best- and worst-case complexities.

In the following, $s, s1, \dots$ are sequences and $i, i1, \dots$ are integers.

Operation	English/math	Python	Complexity
length	$ s $	<code>len(s)</code>	$\Theta(1)$
membership	$item \text{ in } s \text{ or } item \in s$	<code>item in s</code>	$\Theta(1), \Theta(s)$
minimum, maximum	$\min(s) \max(s)$	<code>min(s) max(s)</code>	$\Theta(s)$
comparisons	$s1 = s2, s1 < s2, s1 == s2, \text{etc.}$ etc.		$\Theta(1), \Theta(\min(s1 , s2))$
concatenation	$s1 + s2$	<code>s1 + s2</code>	$\Theta(s1 + s2)$
repeated concatenation	$s \times i$ or $i \times s$	<code>s * i</code>	$\Theta(s \times i)$
indexing	s_i or $s[i]$	<code>s[i]</code>	$\Theta(1)$
slicing	$s[i1:i2]$	<code>s[i1:i2]</code>	$\Theta(i2 - i1)$
sorting	s in ascending order s in descending order	<code>sorted(s)</code> <code>sorted(s, reverse=True)</code>	$\Theta(s), \Theta(s ^2)$ $\Theta(s), \Theta(s ^2)$

The indexing operation obtains the item at the given index. The first (left-most) item is at index zero.

The slice $s[i1:i2]$ is the sequence from $s[i1]$ to $s[i2-1]$. In Python, either index can be omitted: $s[:i]$ is the same as $s[0:i]$ and $s[i:]$ is the same as $s[i:len(s)]$.

If sequences $s1, s2$ and s satisfy $s1 + s2 = s$, then $s1$ is a **prefix** of s and $s2$ is a **suffix** of s . If sequences $s1, s2, s3$ and s satisfy $s1 + s2 + s3 = s$, then $s2$ is a **substring** of s .

The following operations apply to mutable sequences only.

Operation	English	Python	Complexity
replace item	let $s[i]$ be <i>new</i>	<code>s[i] = new</code>	$\Theta(1)$
remove item	remove $s[i]$	<code>s.pop(i)</code>	$\Theta(s - i)$
insert item	insert <i>new</i> at i in s	<code>s.insert(i, new)</code>	$\Theta(s - i)$
append item	append <i>new</i> to s	<code>s.append(new)</code>	$\Theta(1)$
sort sequence	put s in ascending order put s in descending order	<code>s.sort()</code> <code>s.sort(reverse=True)</code>	$\Theta(s ^2)$ $\Theta(s ^2)$

Sequence $s1$ is a **subsequence** of s if $s1$ can be obtained from s by deleting zero or more, not necessarily consecutive, items.

4.9.2 IPython

The IPython command `%run -i filename` runs the code in file `filename.py`, which must be in the same folder as the notebook. If the file is in a different folder, write `%run -i path/filename`. We will use this to load auxiliary files with repeatedly used code. Don't modify the `m269_...py` files in folder `notebooks`.

4.9.3 Python

The `from m import f` statement imports function `f` from module `m`. This allows subsequent code to refer directly to `f`, rather than using `m.f`.

A **constant** is a variable that keeps its initial value. Names of constants are written in uppercase.

A **constructor** is a function with the same name as a type. It creates values of that type.

Trying to apply an operation to the wrong type of values leads to a type error. Trying to access an item outside the range of indices leads to an index error. Python supports negative indices, which access items from right to left: the last (right-most) item is at index `-1`.

If a function body ends without executing a return statement, then the return value is `None`, a keyword that represents 'nothing'. The only operations on `None` are the equality and inequality comparisons.

A **method** is a function that is only known in the context of a data type. It is called using **dot notation**: `expression.method(arguments)`. To consult the documentation of a method, use dot notation: `help(type.method)`.

Strings

A string literal starts and ends with the same kind of quote marks, either single quote (`'`), double quote (`"`) or three quotes (`'''` or `"""`). The enclosing quotes are not part of the string, and they cannot occur in the string, e.g. a string enclosed in single quotes must not contain single quotes. Strings spanning multiple lines must be enclosed in three quotes.

The `str` constructor produces a string representation of Booleans, numbers, lists and tuples. The `int` constructor converts a string to an integer.

In Python, `s1 in s2` checks if `s1` is a substring of `s2`. This corresponds to the membership operation if `s1` is a single character.

Ranges

The data type `range` represents an immutable sequence of integers. The expression `range(start, end, step)` is the sequence `start, start+step, ..., until end-1`.

Lists

A list literal is enclosed by square brackets, with items separated by commas. The `list` constructor converts any sequence type to a list.

Tuples

Tuple literals are enclosed in parentheses, with items separated by commas. A tuple of length one is written `(item,)`.

Iteration

English	Python
for each <i>i</i> from <i>i1</i> to <i>i2</i> :	<code>for i in range(i1, i2+1):</code>
for each <i>i</i> from <i>i1</i> down to <i>i2</i> :	<code>for i in range(i1, i2-1, -1):</code>
for each <i>item</i> in <i>sequence</i> :	<code>for item in sequence:</code>
while <i>condition</i> :	<code>while condition:</code>
repeat <i>n</i> times:	<code>for times in range(n):</code>

A repeat-until loop like

1. repeat:
 1. do something
 2. until *condition*

is translated to

```
stop = False
while not stop:
    # do something
    stop = condition
```

Both `for` and `while` are keywords.

A for-loop and ‘repeat *n* times’ should be used when the number of iterations is known before entering the loop; a while-loop or repeat-until loop should be used when the algorithm must decide each iteration whether to continue or stop, respectively. A while-loop may execute zero or more times; a repeat-until loop is executed one or more times.

The following are equivalent ways of going through the items in a sequence.

```
index = 0
while index < len(sequence):
    item = sequence[index]
    # process item
    index = index + 1
```

```
for index in range(len(sequence)):
    item = sequence[index]
    # process item
```

```
for item in sequence:
    # process item
```

A nested loop is a loop within another, e.g. to go through all cells of a table.

4.9.4 Problems

A **search problem** requires finding one or more items in a sequence that satisfy one or more conditions. A **linear search** checks every item of the sequence one by one. A **global condition**, that has to be satisfied by the whole sequence, can be represented by a **Boolean flag** that's set when the condition is satisfied.

To define an operation that modifies some of its inputs, use this template:

Operation: name

Inputs/Outputs: variables that are modified

Inputs: inputs that aren't modified

Preconditions: conditions on the inputs and inputs/outputs

Output: output that isn't an input

Postconditions: conditions relating the inputs to the outputs

In the postconditions, pre- x is the value of input/output variable x before the operation and post- x is its value after the operation.

An **in-place algorithm** works directly on the input/output sequence, without using an additional sequence.

The object ADT consists of all values of all other ADTs and two operations: equality and inequality.

4.9.5 Testing

In M269, we write, check and run test tables in Python as follows:

```
from algoesup import check_tests, test

problem_name_tests = [
    # case,      input1,      input2,      ..., output
    ("test 1",  value1_1,   value2_1,   ..., output_1),
    ("test 2",  value1_2,   value2_2,   ..., output_2),
    ...
]

check_tests(problem_name_tests, [input1_type, input2_type, ...,
                                ↴output_type])

def function_name(input1: input1_type, input2: input2_type, ...) ->_
    ↴output_type:
    ...

test(function_name, problem_name_tests)
```

4.9.6 Complexity

A best- or worst-case scenario is a group of problem instances of varying sizes for which the operation takes the least (respectively, most) time to execute.

A **quadratic-time algorithm** has complexity $\Theta(n^2)$, where n is an integer expression, e.g. the length of the input sequence. The algorithm's run-time quadruples when the input doubles.

CHAPTER 5

TMA 01 PART 1

This study-free week is for you to catch up if you need to, and to complete the first part of TMA 01. If you haven't done so yet, download TMA 01 from the '[Assessment](#)' tab of the M269 website, and put it in a `TMA01` subfolder of your M269 folder.

There's no recipe to solve a computational problem, but if you ask yourself the right questions then you're more likely to find a solution without multiple trials and errors. Sections 1 and 2 of this chapter provide a non-comprehensive checklist, often in question form, of things to think about in order to solve a problem, be it an exercise or TMA question.

M269 uses a very small subset of the Python language. This is to help you write simple algorithms that are easier to understand, analyse and debug than if using the full range of Python operations. Using a small set of operations doesn't necessarily make the algorithms long.

Unless a TMA question explicitly states otherwise, your code can only use the Python subset introduced in this book; otherwise, you may lose some or all marks. Section 3 of this chapter explains how adherence to the taught subset is partially checked.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

5.1 Problem definition

The first stage of the process is to read the informal problem description and translate it into a succinct but precise problem definition. The first question to ask yourself is:

- What type of problem is it? Is it asking to
 - check a property of the input? (decision problem)
 - assign a category to each input? (classification problem)
 - find one or more items satisfying some criteria? (search problem)

Remember that a search problem may not be stated as such.

M269 problems are operations on some data, so the next question is:

- Is the operation a function in the mathematical sense, i.e. does it not modify its inputs?

If it is, fill in this template:

Function: the name of the function

Inputs: the name and type of each input variable

Preconditions: any conditions on the inputs

Output: the name and type of the output variable

Postconditions: how the output relates to the inputs

If it isn't, fill in this one instead:

Operation: the name of the operation

Inputs/Outputs: the name and type of each variable that is modified

Inputs: the name and type of each input variable

Preconditions: any conditions on the inputs

Output: the name and type of the output variable

Postconditions: how the outputs relate to the inputs

In the second template, use pre- x and post- x in the postconditions to refer to the value of input/output variable x before and after the operation, respectively.

Ask yourself the following questions to fill out the templates.

5.1.1 Problem and output names

The problem name states what the function or operation does or produces. The names of the problem and output variable are often similar or related.

- What does the function/operation do? What does it produce?
- If it's a decision problem, what yes/no question does it ask of the input?
- If it's a search problem, what is being looked for?

5.1.2 Inputs and outputs

M269 problems usually have at most one input/output variable, which must be a mutable sequence, and exactly one output-only variable.

- In the problem description, what is given (input), what is asked for (output) and what changes (input/output)?
- Thinking backwards, what data is needed to compute the function's output?
- If it's a decision problem then the output is a Boolean, unless the description says otherwise. What does it represent when it's true?
- If it's a search problem, what kind of input sequence is searched?
- If it's a categorisation problem, what are the possible categories?

- For each variable, what is its type?

If the value is ...	then use type ...
a number	integer or real number
a logical value	Boolean
a text	string
any sequence	sequence
an immutable sequence	tuple
a mutable sequence	list
any value	object

5.1.3 Preconditions

Not all of the following questions apply to every problem, but they help you not miss some typical preconditions.

- If the input is an integer:
 - What are its smallest and largest values?
 - Can it be zero? Can it only be positive, or negative, or odd?
 - Does it have a unit?
- If the input is a sequence:
 - Can it be empty? Does it have a minimum or maximum length?
 - Is it sorted? By which criterion? Is the order ascending or descending?
 - Must the items be pairwise comparable? This may be necessary for search problems, e.g. to find the largest item.
 - Are items unique or can there be duplicates?
 - What are the preconditions on the items? For example:
 - * If the input is a list of integers, must they be positive?
 - * If the input is a string, can it only contain certain characters?
- Do the preconditions include output-only variables? If so, something's wrong because preconditions only restrict the inputs.

5.1.4 Postconditions

- If it's a decision problem, under which conditions is the output true?
- If it's a classification problem then, for each category, what are the conditions for the input to belong to that category?
- If it's a search problem, what are the search criteria?
- If an input is a sequence:

- What happens if it's empty?
- What happens if it has odd length? What if it has even length?
- If the output is a sequence:
 - Can it be empty? Does it have a minimum or maximum length?
 - Does it have to be a subsequence of an input sequence?
 - Does it have to be sorted?
 - What are the postconditions for its items?
- Does every input occur in the postconditions? If not, the postconditions are still incomplete or the input can be removed, as it's not necessary to compute the output.

5.1.5 Test table

Each test is a problem instance and its expected output, so depending on the pre- and postconditions, some of these questions may not apply, e.g. there may not be a largest input or output, or the input sequence can't have items of different types.

- Does the test table have one column for the test case description, one column per input and one column for the output?
- Do the inputs of each test satisfy the preconditions? If not, you have to revise the test or the preconditions.
- Do the outputs of each test satisfy the postconditions? If not, you have to revise the test or the postconditions.
- Are there tests for the smallest and for the largest possible inputs?
- Are there tests for the smallest and for the largest possible output?
- For an input sequence, are there tests both for even and odd lengths, for different types of items, for sorted and unsorted sequences, and for sequences with no, some or all items of the same value?
- For a classification problem, is there a test for each category boundary?
- For a decision problem, are there tests with true and false outputs?
- For a search problem, are there tests for no, one, multiple and all items matching the search criteria?

5.2 Algorithms and complexity

Designing an algorithm is the most creative part of the problem-solving process. There's no recipe but there are some templates for recurring kinds of problems.

5.2.1 Linear search

An algorithm executes a linear search over a sequence by going through the sequence, item by item. Here are some variations. They aren't algorithms but rather algorithmic templates, or algorithmic patterns, as I called them in TM112 Blocks 1 and 2.

To count how many items satisfy the conditions:

1. let *counter* be 0
2. for each *item* in *sequence*:
 1. if *item* satisfies the conditions:
 1. let *counter* be *counter* + 1

To find the first item that satisfies the conditions:

1. let *found* be some value that can't be in the sequence
2. for each *item* in *sequence*:
 1. if *item* satisfies the conditions:
 1. let *found* be *item*
 2. stop

To find all items that satisfy some conditions:

1. let *found* be the empty sequence
2. for each *item* in *sequence*:
 1. if *item* satisfies the conditions:
 1. append *item* to *found*

To find the best item in a non-empty sequence:

1. let *best* be *sequence*[0]
2. for each *item* in *sequence*:
 1. if *item* is better than *best*:
 1. let *best* be *item*



Info: These are the M269 versions of Patterns 2.5 (counting), 2.7 (find value), 2.3 (list filtering) and 2.8 (find best value) in TM112 Block 2.

To check if a sequence satisfies several conditions:

1. let *condition 1* be false
2. let *condition 2* be false
3. etc. (set one more flag per condition)

4. for each *item* in *sequence*:
 1. if *item* satisfies the first condition:
 1. let *condition 1* be true
 2. if *item* satisfies the second condition:
 1. let *condition 2* be true
 3. etc.
5. let *is valid* be *condition 1* and *condition 2* and ...

Further variations are possible, like finding the last item that satisfies the conditions, finding the indices instead of the items, etc.

5.2.2 Complexity

When analysing an algorithm:

- Determine the complexity of each step.
- Ignore constant-time steps, except stop and if-statements, because they don't affect the algorithm's complexity.
- If there's at least one if or stop statement, think of which inputs lead to the least or the most work being done and ignore steps that aren't executed for such inputs.
- The complexity of a loop is the product of the number of iterations and the complexity of its body.
- Best- and worst-case scenarios are about the form of the inputs, e.g. the sought item is at the start or the end of a sequence, not their sizes. The smallest and largest inputs are, by themselves, not a best- or worst-case scenario, but they can be part of it.
- The complexity $\Theta(e)$ is stated with an integer expression e over the input and input/output variables.
- Remember that a linear search algorithm over sequence s often has best-case complexity $\Theta(1)$ and worst-case complexity $\Theta(|s|)$.

5.3 Coding style

General:

- Use descriptive names. Avoid abbreviations and single-letter names.
- Use 4 spaces for each indentation level.
- Keep lines shorter than 80 characters, to comfortably fit within the screen.
- Write docstrings and multi-line strings between three double quotes.

Variables:

- Write variable names in lowercase, with underscores separating words, e.g. `test_case`.

- Write constant names in uppercase, with underscores separating words, e.g. TEST_CASE.

Functions:

- Write function names in lowercase, with underscores separating words.
- A function name should describe *what* the function returns or does, not *how* it works, unless you have several functions solving the same problem.
- Indicate the type of each parameter and the output, with type annotations.
- Write a docstring for each function.
- If you're implementing a function in the mathematical sense, make sure your Python function doesn't modify any input.



Info: This and later coding style guidance is partly based on [Style Guide for Python Code](#) (better known as PEP 8) and [Docstring Conventions](#) (PEP 257).

5.3.1 The Zen of Python

The following is about the design of the Python language, but we should strive for the same aims of simplicity, clarity and readability when designing, implementing, documenting and testing our algorithms.

[1]: `import this`

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```



Info: Tim Peters contributed to the design and implementation of Python. The creator of Python, Guido van Rossum, is Dutch.

5.3.2 Linters

Many software companies expect their developers to follow the same coding style, as it makes it easier to understand each others' code and to onboard new developers. Professional developers tend to use tools that automatically check their code. Such tools are called linters.

One of the best linters for Python is Ruff, which is included in the M269 software. All TMA notebooks have code like this:

```
[2]: %load_ext algoesup.magics
%ruff on --extend-ignore E711
ruff was activated
```

After this code is executed, each time you run a code cell, it is automatically linted, like this:

```
[3]: def length(list):
    count = 0
    for value in list:
        count = count + 1
ruff found issues:
```

- 1: [ANN201] Missing return type annotation for public function `length`. Suggested fix:
Add return type annotation: `None`
- 1: [D103] Missing docstring in public function
- 1: [A002] Function argument `list` is shadowing a Python builtin
- 1: [ANN001] Missing type annotation for function argument `list`
- 3: [B007] Loop control variable `value` not used within loop body. Suggested fix:
Rename unused `value` to `_value`

Ruff reports the line and number of each issue, e.g. line 3 has issue B007. Clicking on the issue number takes you to Ruff's manual for further explanation, e.g. click on A002 to understand what 'shadowing a Python builtin' means.

Note that Ruff also detects the absence of `return` because it suggests that the output type should be `None`. Ruff has no error code for a missing `return` because it can't guess whether the function is supposed to return a value or not.

If Ruff is wrongly flagging an issue, you can tell Ruff to ignore it by ending the 'offending' line with the comment `# noqa: ...` where '...' is the error code. For example, if a loop variable isn't used in the loop's body, Ruff flags error B007. But sometimes, as in the code above, the loop variable is *not* meant to be used, and Ruff suggests a fix based on a Python convention we don't

use in M269. So in this case you should add a ‘no quality assurance’ (`noqa`) comment for Ruff to ignore the issue. Here’s the new version of the code, addressing all the issues found by Ruff.

```
[4]: def length(values: list) -> int:
    """Return the number of items in the list."""
    count = 0
    for value in values: # noqa: B007
        count = count + 1
    return count
```

You will find throughout the book several `noqa` comments. The most frequent ones are for Ruff to ignore missing docstrings, because I omit them when it’s a new version of a function I wrote before, so that you can concentrate on the code changes.

While we suggest you keep code lines shorter than 80 characters, Ruff only flags lines longer than 88 characters to give some leeway for longer descriptive names and for functions with several parameters.



Note: Ruff may flag issues or make suggestions that are not appropriate in an educational setting like M269. When in doubt about whether to address a Ruff message, post its error code in the Technical Forum.

All TMA notebooks also contain code like this:

```
[5]: import platform

[6]: if platform.system() in ("Linux", "Darwin"):
    %allowed on --config m269-25j --unit 5 --method
else:
    %allowed on --config m269-25j --unit 5
allowed was activated
```

This makes the `allowed` linter, also included in the M269 software, check your code against the Python subset taught until chapter 5 of this book. On Linux and macOS, `allowed` will also check some method calls. Here’s an example:

```
[7]: from math import sqrt

def some_function(n: int) -> int:
    """To be implemented."""
    arabic_to_roman = {1: "I", 5: "V"}
    numbers = [1, 2, 1, 0]
    numbers.count(1)
    pass
```

allowed found issues:

- 1: sqrt
- 6: dict literal
- 8: list.count()
- 9: pass

ruff found issues:

- 6: [F841] Local variable `arabic_to_roman` is assigned to but never used. Suggested fix:
Remove assignment to unused variable `arabic_to_roman`

The `sqrt` function, the dictionary literal and the `pass` statement are flagged because they aren't taught in Chapters 1 to 5. (They will be introduced later.) The `count` method on lists, which isn't used in M269, is also flagged because I use macOS.



Note: If you are on Windows, you can check method calls by uploading your TMA to the Open Computing Lab and running all cells there.



Note: Linters aren't perfect. Even if they don't flag anything, your code may have issues: syntax errors, disallowed constructs, etc. Always read your code carefully before submitting your TMA.



Info: The `allowed` linter was co-written by former M269 students. If you're interested in contributing to the tool's development, see its [website](#).

CHAPTER 6

IMPLEMENTING SEQUENCES

This chapter explains how the sequence ADT can be implemented using two different **data structures**, i.e. ways of organising data: arrays and linked lists. A data type implements an ADT using a particular data structure. For example, Python's list data type implements the sequence ADT with arrays, but we can define our own data type implementing the same ADT with a linked list.

The same data structure can be used by data types implementing different ADTs. Chapter 7 introduces other sequence-like ADTs and how they can be implemented with the array and linked list data structures.



Info: Some texts use 'data structure' as a synonym of 'data type', i.e. as the implementation of an abstract data type.

Arrays and linked lists are **linear** data structures: the data is arranged sequentially. This makes it easy to go through the data items, one by one, with a single loop. Linear data structures are widely used by Python and other languages to implement various kinds of sequence ADTs.

Learning about data structures not only helps you understand a data type's behaviour, e.g. why some operations take constant time, it also gives you the foundations to implement ADTs.

This chapter supports the following learning outcomes.

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn two fundamental data structures (arrays and linked lists).
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – this chapter illustrates how to communicate the gist of an algorithm or data structure.
- Analyse the complexity of algorithms to support software design choices – this chapter shows how the choice of data structure affects the complexity of the data type's operations.

- Write readable, tested, documented and efficient Python code – this chapter shows how to implement data types with Python classes and how to test them.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

6.1 Defining data types

Before looking at how to implement the sequence ADT, let's start with a simpler example that illustrates the difference between a data structure and a data type, and how to define a data type in Python.

Let's suppose we want to define and implement an ADT for fractions $\frac{x}{y}$, where x and y are integers, with x being the numerator and $y \neq 0$ being the denominator.



Info: MU123 Unit 3 Section 2 introduces fractions and their operations.

Programming languages have special syntax for literals of built-in data types, like integers and strings and other sequence types. However, for our own types we need to define constructor operations that create values of the new type from values of other types, e.g. like Python's constructor to create strings from integers. To highlight such operations, we shall use 'Constructor' instead of 'Function' in the template. Here's how two of the fraction ADT's operations could be defined.

ADT: fraction

Constructor: new fraction

Inputs: *numerator*, an integer; *denominator*, an integer

Preconditions: *denominator* $\neq 0$

Output: *ratio*, a fraction

Postconditions: $\text{ratio} = \frac{\text{numerator}}{\text{denominator}}$

Function: multiplication

Inputs: *left*, a fraction; *right*, a fraction

Preconditions: true

Output: *product*, a fraction

Postconditions: if $\text{left} = \frac{ln}{ld}$ and $\text{right} = \frac{rn}{rd}$, then $\text{product} = \frac{ln \times rn}{ld \times rd}$

Let's see how to implement the fraction ADT in Python.

6.1.1 Data structure

The first step in implementing an ADT is to choose how to structure the data. A fraction is represented by two integers, so the obvious choice is to use a tuple or list with a pair of integers. A tuple is a better choice, to prevent changes to the numerator or denominator. Furthermore, we must state what each integer represents. It's probably more intuitive for the first integer of the tuple to be the numerator and the second to be the denominator.

Data structures vary widely and we can't fit their description into a template, like we do for operations. During your professional life you may need to explain what data structures you use, verbally to a colleague, or in writing documentation, and so it's important you communicate the structure of data in plain but clear English, or whatever language you may use at work. For this example, a good description would be:

The data structure to represent a fraction is a tuple of two integers, the first being the numerator and the second the non-zero denominator.

6.1.2 Functions

Having decided the data structure, we can implement the operations. Until now we implemented each operation with a Python function, like the following. (I omit docstrings to keep this temporary solution short.)

```
[1]: def fraction(numerator: int, denominator: int) -> tuple: # noqa: D103
    return (numerator, denominator)

def multiplication(left: tuple, right: tuple) -> tuple: # noqa: D103
    return (left[0] * right[0], left[1] * right[1])
```

We use the constructor operation to create new values, which are used by the other operations.

```
[2]: half = fraction(1, 2)
multiplication(half, half) # one half times one half is a quarter
[2]: (1, 4)
```

This way of implementing ADTs is unsatisfactory because it exposes the data structure to the user. This allows the user to bypass the constructor and make (by mistake) calls like

```
[3]: multiplication((1, 2), (1, 3, 5))
[3]: (1, 6)
```

where the second argument doesn't represent a fraction. Moreover, if we change the data structure then the user will likely have to change their code. We need a better approach.

6.1.3 Classes

A data type is a collection of values and operations on those values. Python and other languages have a construct to bundle together data and functionality: **classes**. Each class implements a data type: the values are the **instances** of the class and the operations are the class's **methods**. An **object** is an instance of some class, and that's why I used the word 'object' in templates to denote any value. For example, 5 and [] are objects: 5 is an instance of class `int` and [] is an instance of class `list`, which has methods like `pop` and `append`. Methods are usually directly called using the dot notation, but some are indirectly called via operators, like + for addition or concatenation.



Info: Object-oriented programming is an approach that models a software system as a collection of objects calling each other's methods. M250 explains this approach at length.

Every instance has some variables, called **instance variables**, to hold the data for that instance. When defining a class we define the variables for all instances of that class. Different instances typically have different values for their variables. Let me show you how to define a class in Python. I explain the code afterwards.

```
[4]: class Fraction:
    """A ratio represented as a pair of integers: numerator and non-
    zero denominator."""

    def __init__(self, numerator: int, denominator: int) -> None:
        """Initialise the fraction.

        Preconditions: denominator != 0
        """
        self.value = (numerator, denominator)

    def multiplication(self, right: "Fraction") -> "Fraction":
        """Return the product of self and right.

        Postconditions: if self is the fraction sn/sd and
        right is the fraction rn/rd, then the output is fraction
        (sn*rn) / (sd*rd)
        """
        numerator = self.value[0] * right.value[0]
        denominator = self.value[1] * right.value[1]
        return Fraction(numerator, denominator)
```

The definition of a class C starts with `class C:` and a docstring describing the data type being defined, followed by the methods, which are defined like any Python function. All methods are indented: they are 'within' the class. The name of a class reflects what each instance represents, so it's usually in the singular. The names of built-in classes are usually in lowercase, but the names of classes we define should use capitalised words without underscores to separate them,

e.g. `FractionNumber`.

In M269, each class must have a method named `__init__`, with two underscores at the start and at the end. This method takes as first argument an instance of the class, conventionally called `self`, and possibly additional arguments with data to initialise that instance: here, the numerator and denominator of the fraction. The body of the method creates the instance variables, using dot notation to indicate that these variables ‘belong’ to the instance. For every instance variable `x` there’s an assignment to `self.x`. In this example, I create an instance variable `value` that holds the tuple with the two integers passed to the method.

Each built-in class, like `range` and `list`, has a constructor: a function with the same name as the class to create instances of that class. The Python interpreter automatically defines a constructor for every class we define, with the same arguments as method `__init__`, except for `self`. For this example, the interpreter creates a constructor `Fraction` with two arguments, which we use to create new fractions.

```
[5]: one_half = Fraction(1, 2)
```

When we call the constructor, the interpreter creates an ‘empty’ instance, without any data, and passes it as the first argument to `__init__`, which creates the instance variables and assigns values to them. The constructor (not the `__init__` method!) then returns the instance: that’s why there are no return statements in the `__init__` method.

We can use the `help` function to obtain information about any class. Some of the information is of no relevance to M269.

```
[6]: help(Fraction)

Help on class Fraction in module __main__:

class Fraction(builtins.object)
|   Fraction(numerator: int, denominator: int) -> None
|
|   A ratio represented as a pair of integers: numerator and non-
→zero denominator.
|
|   Methods defined here:
|
|   __init__(self, numerator: int, denominator: int) -> None
|       Initialise the fraction.
|
|       Preconditions: denominator != 0
|
|   multiplication(self, right: 'Fraction') -> 'Fraction'
|       Return the product of self and right.
|
|       Postconditions: if self is the fraction sn/sd and
|       right is the fraction rn/rd, then the output is fraction
|       (sn*rn) / (sd*rd)
|
```

(continues on next page)

(continued from previous page)

```
| -----
| Data descriptors defined here:
|
|     __dict__
|         dictionary for instance variables
|
|     __weakref__
|         list of weak references to the object
```

Note that the `help` function copies the header of the `__init__` method for the constructor. This gives the erroneous impression that the constructor doesn't return anything.

Methods are called with dot notation. The interpreter looks up the class of the object to the left of the dot and calls the method defined in that class. The first argument of every method is therefore an instance of the class being defined. We conventionally call it `self` and don't indicate explicitly its type, which is `Fraction` in this example. Here's an example of calling a method.

```
[7]: one_half.multiplication(one_half)
```

```
[7]: <__main__.Fraction at 0x1085480b0>
```

Unfortunately, the output message is not very useful. The interpreter shows the class and unique id of the resulting object. The id is an integer, shown here in hexadecimal notation. Printing the fraction doesn't help either.

```
[8]: print(one_half.multiplication(one_half))
<__main__.Fraction object at 0x108548470>
```

If we want to see the value of an instance in a meaningful way, we have to implement a method `__str__` (again, two underscores before and after) that returns a string. The `print` function calls the `str` constructor on each object it prints, which in turn calls the `__str__` method on that object.

Here's the class again, with the additional method.

```
[9]: class Fraction:
    """A ratio represented as a pair of integers: numerator and non-
    zero denominator."""

    def __init__(self, numerator: int, denominator: int) -> None:
        """Initialise the fraction.

        Preconditions: denominator != 0
        """
        self.value = (numerator, denominator)
```

(continues on next page)

(continued from previous page)

```

def multiplication(self, right: "Fraction") -> "Fraction":
    """Return the product of self and right.

    Postconditions: if self is the fraction sn/sd and
    right is the fraction rn/rd, then the output is fraction
    (sn*rn) / (sd * rd)
    """
    numerator = self.value[0] * right.value[0]
    denominator = self.value[1] * right.value[1]
    return Fraction(numerator, denominator)

def __str__(self) -> str:
    """Return a string representation of the fraction."""
    return str(self.value[0]) + " / " + str(self.value[1])

```

Now we can see the value of a fraction:

```
[10]: ONE_HALF = Fraction(1, 2)
ONE_QUARTER = ONE_HALF.multiplication(ONE_HALF)
ONE_QUARTER # display class and unique id
[10]: <__main__.Fraction at 0x10850d4c0>
```

```
[11]: print(ONE_QUARTER)
str(ONE_QUARTER)
1 / 4
[11]: '1 / 4'
```

6.1.4 Mistakes

Whenever you change a class you must rerun code cells that create instances: otherwise they remain instances of the old version of the class. Consider this:

```
[12]: print(half) # created with fraction(1, 2); instance of tuple
print(one_half) # created with first version of Fraction class
print(ONE_HALF) # created with second version of Fraction class
(1, 2)
<__main__.Fraction object at 0x10850ffb0>
1 / 2
```

Unless you have rerun cells in a different order, the middle output of the cell above isn't '1 / 2', because `one_half` was created with the constructor for the class without the `__str__` method.

The name of a class becomes known only *after* processing the class definition. If we need to use the class name in the header of a method, then we must write it as a string, as I've done for the `multiplication` method. If you forget the string quotes you get a name error.

```
[13]: class Date: # noqa: D101
    # docstrings and __init__ omitted to focus on the issue at hand

    def difference(self, other: Date) -> int:
        """Return number of days between two dates."""
        return 0 # dummy code

-----
↔-----  
NameError Traceback (most recent)
→call last)
Cell In[13], line 1
----> 1 class Date: # noqa: D101
  2_
→ 3     # docstrings and __init__ omitted to focus on the issue at hand
  4     def difference(self, other: Date) -> int:
  5         """Return number of days between two dates."""

Cell In[13], line 4, in Date()
  1 class Date: # noqa: D101
  2     # docstrings and __init__ omitted to focus on the issue
→at hand
----> 4     def difference(self, other: Date) -> int:
  5         """Return number of days between two dates."""
  6         return 0

NameError: name 'Date' is not defined
```

As explained in [Section 4.6.4](#), method names are only known in the context of their class. Calling a method as if it were a standalone function usually raises a name error.

```
[14]: __str__(ONE_QUARTER)

-----
↔-----  
NameError Traceback (most recent)
→call last)
Cell In[14], line 1
----> 1 __str__(ONE_QUARTER)

NameError: name '__str__' is not defined
```

```
[15]: ONE_QUARTER.__str__()

[15]: '1 / 4'
```

However, if a standalone function of the same name exists, the interpreter will call that one.

```
[16]: multiplication(ONE_HALF, ONE_HALF)

-----
→-----  

TypeError Traceback (most recent call last)
Cell In[16], line 1
----> 1 multiplication(ONE_HALF, ONE_HALF)

Cell In[1], line 6, in multiplication(left, right)
      5 def multiplication(left: tuple, right: tuple) -> tuple: #_
      6     ↪noqa: D103
----> 6     return (left[0] * right[0], left[1] * right[1])

TypeError: 'Fraction' object is not subscriptable
```

We get a type error because the indexing operation is not defined on fractions. The standalone function expects two tuples, not two fractions. `Fraction` and `tuple` are different types, even though `Fraction` has an instance variable of type `tuple`.

Likewise, I cannot pass a tuple to a method expecting a fraction.

```
[17]: one_half.multiplication((1, 2))

-----
→-----  

AttributeError Traceback (most recent call last)
Cell In[17], line 1
----> 1 one_half.multiplication((1, 2))

Cell In[4], line 18, in Fraction.multiplication(self, right)
    11 def multiplication(self, right: "Fraction") -> "Fraction":  

    12     """Return the product of self and right.  

    13  

    14     Postconditions: if self is the fraction sn/sd and  

    15     right is the fraction rn/rd, then the output is fraction  

    16     (sn*rn) / (sd*rd)  

    17     """  

----> 18     numerator = self.value[0] * right.value[0]
    19     denominator = self.value[1] * right.value[1]
    20     return Fraction(numerator, denominator)

AttributeError: 'tuple' object has no attribute 'value'
```

I get a special case of a name error: the interpreter is complaining that tuples don't have an instance variable named `value`. In Python, the instance variables and methods are a class's **attributes**. If you have two attributes with the same name, e.g. an instance variable and a method, you will get errors. In the following incomplete example of a class for dates (day,

month, year), the name `day` refers both to an integer instance variable and a method.

```
[18]: class Date: # noqa: D101
    def __init__(self) -> None: # noqa: D107
        self.day = 1

    def day(self) -> int: # noqa: D102
        return self.day
```

We can access the instance variable...

```
[19]: Date().day
[19]: 1
```

...but not call the method.

```
[20]: Date().day()
-----
↑-----  
TypeError                                     Traceback (most recent...)
←call last)
Cell In[20], line 1
----> 1 Date().day()

TypeError: 'int' object is not callable
```

As the above example shows, Python doesn't prevent users of a class from accessing its instance variables. Here's another example.

```
[21]: if 0 < ONE_HALF.value[0] < ONE_HALF.value[1]:
    print(ONE_HALF, "is positive and smaller than 1")
1 / 2 is positive and smaller than 1
```

This code relies on a particular instance variable name and data structure for representing fractions. If either changes, the code won't work. For example, imagine we replace the tuple with two integer instance variables:

```
[22]: class Fraction: # noqa: D101
    # docstrings omitted to focus on data structure changes

    def __init__(self, numerator: int, denominator: int) -> None: #_
        →noqa: D107
        self.numerator = numerator
        self.denominator = denominator

    def multiplication(self, right: "Fraction") -> "Fraction": #_
        →noqa: D102
```

(continues on next page)

(continued from previous page)

```

numerator = self.numerator * right.numerator
denominator = self.denominator * right.denominator
return Fraction(numerator, denominator)

def __str__(self) -> str: # noqa: D105
    return str(self.numerator) + " / " + str(self.denominator)

print(Fraction(1, 2).multiplication(Fraction(1, 3)))
1 / 6
    
```

Any code that uses the previous version of the class without accessing the instance variables also works with this version, because the **interface** of the class, i.e. its methods and their headers, hasn't changed.



Note: Only the class's methods should access the instance variables.



Info: Unlike Java, Python doesn't have access modifiers to make instance variables private or protected.

Accessing instance variables from outside a class is poor programming practice. If you need access to instance variables to solve a problem, you may not have defined enough methods. For example, the `Fraction` class should provide methods to return the numerator and denominator of a fraction. I'm adding only one of them, to illustrate how methods can call each other. Compare the following version to the previous one.

```
[23]: class Fraction: # noqa: D101
    def __init__(self, numerator: int, denominator: int) -> None: #_
        →noqa: D107
        self.top = numerator
        self.denominator = denominator

    def numerator(self) -> int: # noqa: D102
        return self.top

    def multiplication(self, right: "Fraction") -> "Fraction": #_
        →noqa: D102
        numerator = self.numerator() * right.numerator()
        denominator = self.denominator * right.denominator
        return Fraction(numerator, denominator)
```

(continues on next page)

(continued from previous page)

```
def __str__(self) -> str: # noqa: D105
    return str(self.numerator()) + " / " + str(self.denominator)

print(Fraction(1, 2).multiplication(Fraction(1, 3)))
1 / 6
```

I renamed one instance variable, so that it doesn't have the same name as the new method. Note that `x.y()` calls method `y` on instance `x`, whereas `x.y` accesses variable `y` of instance `x`.

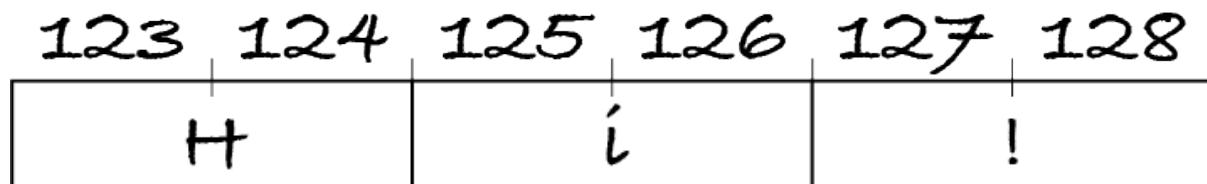
Assume that a second method to obtain the denominator is added. Can you think of an advantage for the multiplication, string conversion and other methods to be added to call the auxiliary methods instead of directly accessing the instance variables?

If the data structure further changes, only the `__init__`, `numerator` and `denominator` methods must change; all other methods on fractions (multiplication, addition, string conversion, rounding, etc.) remain the same.

6.2 Static arrays

A **static array** is a data structure that stores a fixed-length sequence contiguously in memory, with the same number of bytes per item. This allows us to compute the address of any item, and thereby access it, in constant time. For example, the next figure shows a static array storing the string 'Hi!' from memory address 123 onwards, using two bytes per character.

Figure 6.2.1



The character at index i is stored at the address $123 + i \times 2$ and the next address, e.g. the character at index 1 starts at address $123 + 1 \times 2 = 125$. (Remember that indices start at zero.) More generally, if an array at memory $start$ allocates n bytes per item, the item at index i can be found at address $start + i \times n$ and the subsequent $n - 1$ bytes.



Info: The term 'array' can mean different things (static array, list, sequence of items of the same type, etc.), depending on the author or programming language. To avoid confusions, we use 'static array' in M269, with the above meaning.

Static arrays are the ideal data structure for immutable sequences where all items occupy the same number of bytes, like strings and tuples of Booleans. But how to store a tuple of strings of different lengths, or a tuple with items of different types? The trick is to store in the array the addresses of the items instead of the items themselves. Let's suppose we have 2^{32} bytes = 4 GB of RAM. We can represent any memory location as a 32-bit integer. The next figure illustrates the storage of tuple ('Hi!', 'Go', 'left'). The static array for the tuple contains the address of each string. The strings may be anywhere in memory, not necessarily next to the tuple or each other.

Figure 6.2.2

1050..1053	1054..1057	1058..1061
123	100542	1423208

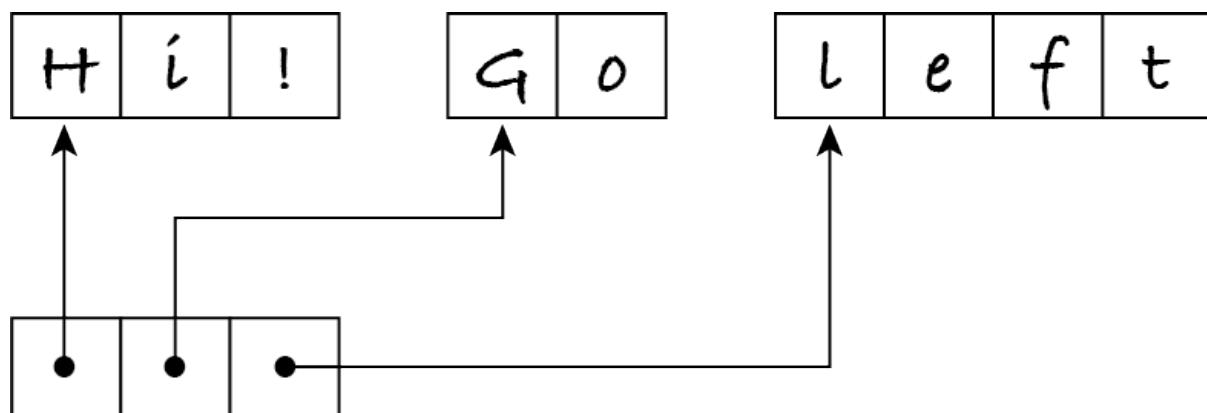
123	124	125	126	127	128
H	i			!	

100542	100543	100544	100545
G		O	

1423208/9	1423210/1	1423212/3	1423214/5
l	e	f	t

We usually don't know (and don't care) where in memory a data structure will be stored. It's also often irrelevant how many bytes characters, addresses and other items occupy. Abstracting away those details, we can represent the tuple and its three strings in a more schematic way.

Figure 6.2.3



This image conveys the gist of the data structure: a string is stored as a static array of characters, and the tuple of strings is stored as a static array of references to those strings. A **reference** is an object that refers to another object. If the reference consists of the memory address of that other

object, we call it a **pointer**.

Before continuing with static arrays, let me illustrate how the use of references explains the behaviour of assignments in Python.

6.2.1 Variables and assignments

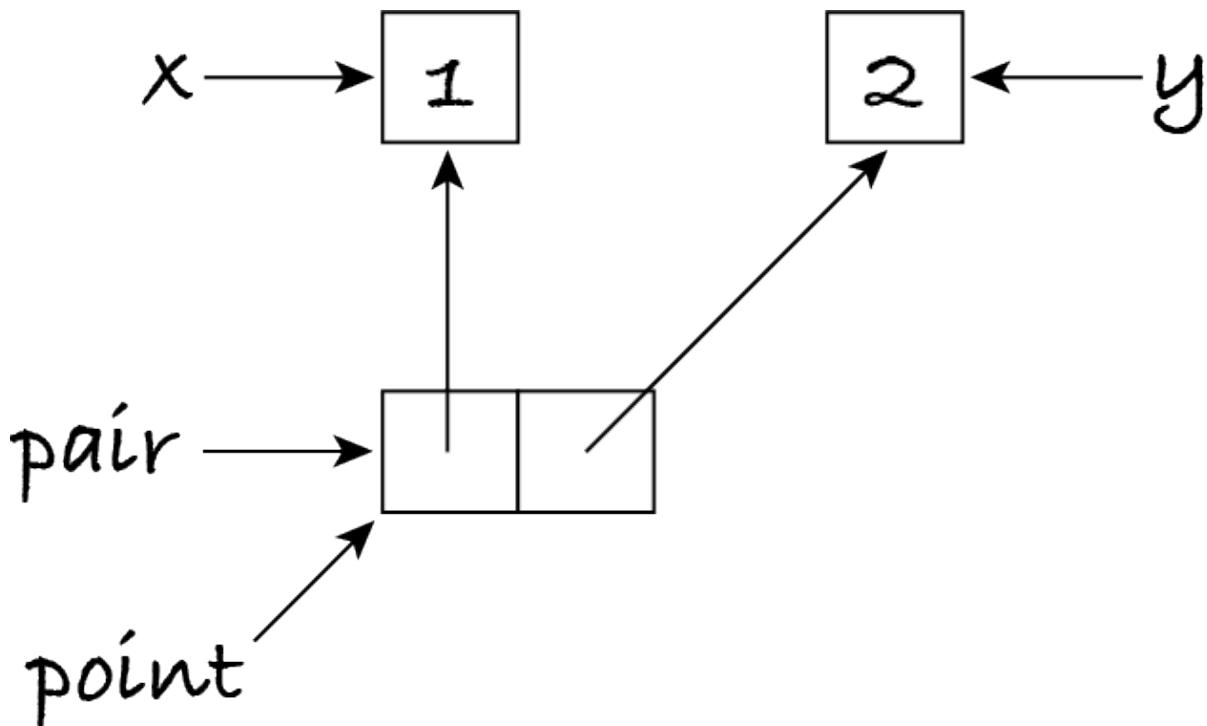
There are other kinds of reference objects. An index is an integer object that refers to a particular item of a sequence. A variable is a named reference to an object: you can think of it as a string–pointer pair with the name of the variable and the address of the object it refers to.

Consider the following example, which uses a list instead of a tuple.

```
[1]: x = 1 # the x-y coordinates of a 2D point  
y = 2  
pair = [x, y]  
point = pair
```

Assuming that lists are also stored as static arrays, the situation in memory at this stage is as follows:

Figure 6.2.4



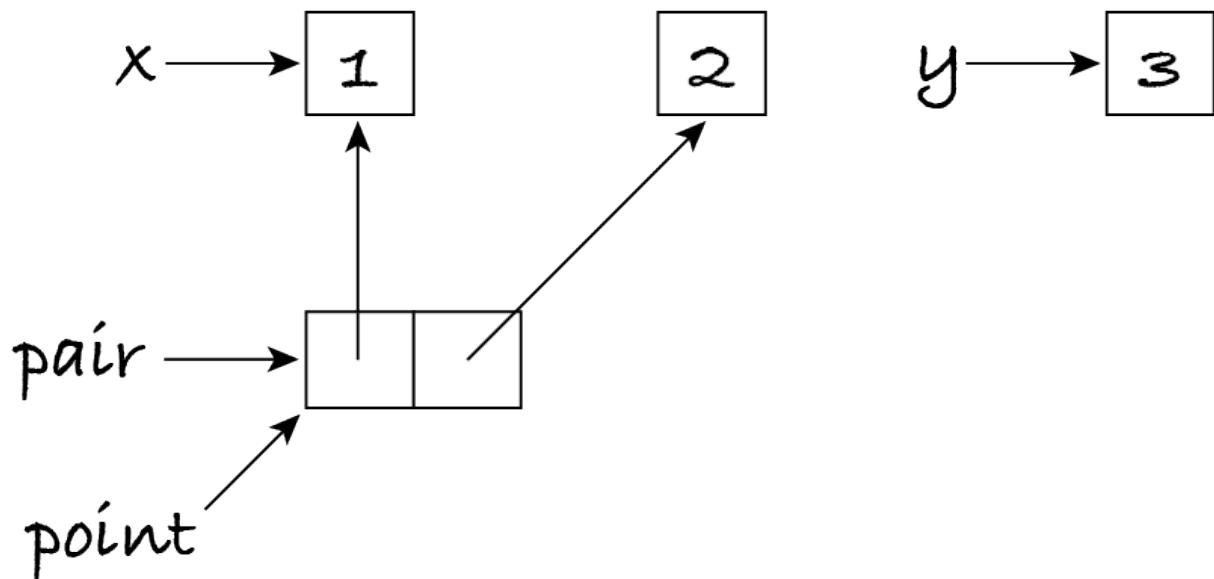
Names `pair` and `point` refer to exactly the same list object, `pair[0]` and `x` refer to the exact same integer object with value one, and `pair[1]` and `y` refer to the same integer object with value two. Let's change the value that `y` refers to:

```
[2]: y = 3
```

The Python interpreter does *not* change the value of the existing integer object. Instead, it creates a new integer object with value three, and makes the variable refer to that object. In

Python, `variable = value` ‘attaches’ the variable’s name to the value, rather than modifies the current variable’s value. The data structures look now like this:

Figure 6.2.5



Hence, if we display the list, it hasn’t changed:

```
[3]: point
```

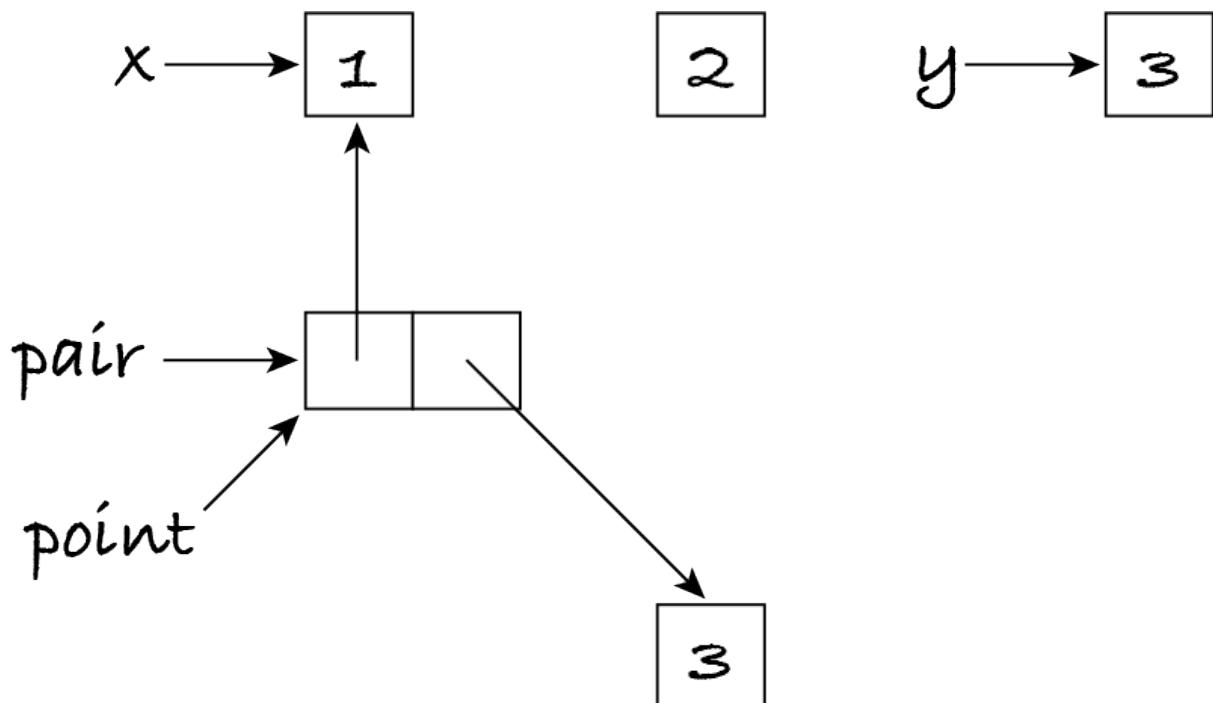
```
[3]: [1, 2]
```

If we do this assignment,

```
[4]: pair[1] = 3
```

then the interpreter again creates an integer object with value 3 and makes the name `pair[1]` refer to that object. The situation is now:

Figure 6.2.6



Let's display again the point's coordinates:

[5]: `point`

[5]: `[1, 3]`

They have changed, even though the assignment was to `pair[1]`, not `point[1]`. The earlier assignment `point = pair` created a new reference to the same list, it didn't create a copy of the list.



Info: TM112 Block 2 Sections 4.5.1 to 4.5.3 explain the above in more detail.

The diagram also shows that the integer object with value two isn't referred by any other object anymore. The Python interpreter detects that automatically, and frees the memory occupied by the object so that it can be reused later. This process is known as **garbage collection**.

6.2.2 The staticArray class

Some languages provide a static array data type to implement a restricted form of the sequence ADT, namely fixed-length sequences of items of the same type. Besides the length of the array, one has to indicate the type of items. This allows the interpreter to figure out how many bytes per item are needed and the initial value of each item, to then create the static array in memory. For example, an array of Booleans may be created with one byte per Boolean, each initialised to false, whereas an array of strings may be created with pointers to the same empty string.



Info: Java has a static array data type, simply called array, with indices starting at zero. The type of items is indicated when the array is declared and the length is determined when the array is initialised. Java arrays can have items of different types if the declared item's type is Object.

A data type for static arrays only has to support three operations, besides the constructor: length, indexing, and replacing an item. All other operations, e.g. slicing, can be implemented with these basic operations.

Python doesn't include such a data type as it already has a far more flexible one: lists. The implementation of lists may differ between Python interpreters, because the Python language only defines the list as an ADT, i.e. which operations it supports. However, it's safe to assume all interpreters use static arrays as the underlying data structure in order to achieve constant-time indexing. We can thus use lists as if they were static arrays providing we never call methods append, insert and pop. I will however define a new data type, to make the use of static arrays explicit in subsequent examples.

I won't restrict my static array data type to any particular type of items. The constructor simply takes the length of the array, and initialises it with `None` in every position. The essential operations on an array are to obtain its length, and to access and replace an item at a given index. Following my definition of the *indexing operation*, I assume indices aren't negative. For convenience I also implement an operation to convert to a string.

I write the class to a file to reuse it later.

```
[6]: # this code is also in m269_array.py

class StaticArray:
    """A fixed-length sequence of references in contiguous memory."""

    def __init__(self, length: int) -> None:
        """Create an array of the given length.

        Preconditions: length >= 0
        Postconditions: every item in the array is None
        """
        # assume lists are stored in arrays
        self.items = [None] * length

    def length(self) -> int:
        """Return the length of the array."""
        return len(self.items)

    def get_item(self, index: int) -> object:
        """Return the item at the given index.

        Preconditions: 0 <= index < self.length()
        """
        pass
```

(continues on next page)

(continued from previous page)

```
"""
    return self.items[index]

def set_item(self, index: int, item: object) -> None:
    """Replace the item at the given index with the given item.

    Preconditions: 0 <= index < self.length()
    Postconditions: self.get_item(index) == item
"""

    self.items[index] = item

def __str__(self) -> str:
    """Return a string representation of the array."""
    return str(self.items)
```

In Python, every object is an instance of a class appropriately named `object`. All other classes are subclasses of `object`. Class *A* is a **subclass** of class *B*, and *B* is a **superclass** of *A*, if every instance of *A* is also an instance of *B*. Using `object` in a method or function header indicates that the input or output value can be any object. As such, the method or function body can only use the equality and inequality comparisons or assign the object to a variable.

6.2.3 Testing methods

Our *generic test function* is meant for testing functions with a single output that can be easily compared against an expected value. However, class methods sometimes modify an instance variable instead of returning a value, like `set_item` above. We need a different approach.

White-box tests, also known as structural tests, have full knowledge of the data structure(s) used by the data type. Structural tests can be written independently for each method, as they can access the instance variables before and after the method to check they were correctly modified. The disadvantage of structural tests is that they must be rewritten whenever the data structures change.

Black-box tests, also known as specification-based tests, don't access the data structures. They test methods based on their defined behaviour. This means that methods can't be tested independently: a method that changes data can only be tested with methods that inspect data. Black-box testing makes **regression testing** easier. That's the process of running the same tests after each change, to make sure the modified code still behaves as before.

For example, a white-box test of the `__init__` method can iterate over the `self.items` list to check all items were initialised to `None`, whereas a black-box test has to use method `get_item`.

In M269 we'll use mostly black-box testing: a single set of tests, based on the ADT's operations, can be run repeatedly to test different implementations of the same ADT. This saves work and time.

To make test code shorter and more readable, I define an auxiliary function for the error messages and write it to a file to reuse it later.

[7]: # this code is also in m269_test.py

```
def check(case: str, actual: object, expected: object, context:_
→object) -> None:
    """Write a message if actual and expected are different."""
    if actual != expected:
        print(case, "FAILED for", context, ":", actual, "instead of",
→ expected)
```

The context is some additional information to help us make sense of the error. In the following, the context is the static array for which the test failed.

Let's start by testing the initialisation method for an arbitrary array length. We must check it has created the right number of items, all of them `None`.

[8]: `def test_init(length: int) -> None:`
 `"""Create a new array of the given length and check it.`

```
    Preconditions: length >= 0
    """
    array = StaticArray(length)
    check("length", array.length(), length, array)
    for index in range(0, length):
        check("initial item", array.get_item(index), None, array)
```

The black-box test uses methods `length` and `get_item` to inspect the created instance. As such, we're also testing these methods besides `__init__`.

We also test `set_item` by creating an array and checking it with `get_item`. I often use the sequence `0, 1, 2, ...` for testing, because if a test fails it's easy to see which value is not at the right index.



Note: When testing operations on sequences, use sequences of the form `(0, 1, 2, ...)`.

[9]: `def test_set_item(length: int) -> None:`
 `"""Create an array of the given length, replace all items and_
→check it.`

```
    Preconditions: length >= 0
    """
    array = StaticArray(length)
    for index in range(length):
        array.set_item(index, index)
    for index in range(length):
```

(continues on next page)

(continued from previous page)

```
check("replaced item", array.get_item(index), index, array)
```

This only tests `set_item` with an array of items of the same type. In general, you may need several test functions for one method. It's not unusual for test code to be longer than the code it tests.

I can now test arrays of different lengths. The `check` function will print every array that fails a test.

```
[10]: test_init()  # edge case: length zero
for length in range(1, 11):
    test_init(length)
    test_set_item(length)
```

In testing, no news is good news.



Note: To black-box test a class, write one or more test functions for each method that creates or modifies an instance of that class, using the methods that inspect the instance. Together, the tests must use all inspection methods.

6.3 Developing data types

In the rest of this chapter I will show you how to implement the sequence ADT with static arrays and other data structures.

A static array cannot grow, so it's convenient to distinguish sequences that are **bounded**, i.e. that have a maximum length by design, from those that aren't. The length of any sequence is limited by the available memory, but bounded sequences have their **capacity** (maximum length) set when they're created. The sequence is full when its length (number of items it has) equals its capacity (number of items it can hold). When a bounded sequence is full, no item can be inserted or appended. A sequence with capacity zero never grows and remains empty. An unbounded sequence can be seen as a sequence with infinite capacity.

Introducing the notion of capacity forces us to change the sequence ADT definition. First, we should add a function to obtain a sequence's capacity. Second, we must change the preconditions of the insertion and append operations: they aren't defined for full sequences. Third, for operations that produce a new sequence, like slicing and concatenation, what should the capacity of the output sequence be? For example, when taking a slice of length s from a sequence with capacity c , should the capacity of the output sequence be s , c or some other value?

It's difficult to decide without knowing how the slice will be used. It's best to add the new capacity as a further input of the operation, so that the user can set a meaningful value for their intended use of the slice. The concatenation operation should also have the capacity of the output sequence as an additional input. Both operations need an additional precondition for the new input: the capacity set by the user must be at least the length of the output, otherwise the

slice or the concatenation won't fit in the new sequence.

To sum up, the plan to implement the sequence ADT with a static array requires the concept of capacity, which leads to a new operation and changes to others.



Note: When implementing a restricted version of an ADT, the definition of some operations may have to change.

6.3.1 Abstract classes

Before we write each implementation of the sequence ADT, it's convenient to define the ADT itself as an **abstract class**: a class that won't be instantiated. The abstract class lists the available methods (operations), but doesn't implement all of them. Each sequence data type will be a subclass of the abstract class, using a particular data structure and implementing the methods. Each sequence object will be an instance of a subclass, not of the abstract class.

In M269 we define an abstract class like any other class, but without an `__init__` method because the abstract class isn't meant to be instantiated. This in turn leads to most methods not being implemented, as there are no instance variables to access. The *Zen of Python* states that explicit is better than implicit, so we'll use Python's `pass` statement to reinforce that the method does nothing.

For the new operation that returns a sequence's capacity, we must somehow represent infinity to handle unbounded sequences. Fortunately, that's easy. The IEEE 754 standard that defines floating-point numbers also includes special values to represent positive and negative infinity. Python's `math` module defines the float constant `inf` for positive infinity. Negative infinity is simply `-inf`.

Here's a partial implementation of the sequence ADT in Python. It follows the definitions of the sequence operations in [Section 4.1](#) and [Section 4.6](#), modified to take the sequence's capacity into account and use Python's syntax.

```
[1]: # this code is also in m269_sequence.py

class Sequence:
    """The sequence ADT."""

    def capacity(self) -> float:
        """Return how many items the sequence can hold.

        Postconditions: if the capacity is only limited by memory,
        the output is math.inf,
        otherwise it's the non-negative integer set at creation time
        """
        pass
```

(continues on next page)

(continued from previous page)

```
def length(self) -> int:
    """Return the number of items in the sequence.

    Postconditions: 0 <= self.length() <= self.capacity()
    """
    pass

def get_item(self, index: int) -> object:
    """Return the item at position index.

    Preconditions: 0 <= index < self.length()
    Postconditions: the output is the n-th item of self, with n
    =>= index + 1
    """
    pass

def set_item(self, index: int, item: object) -> None:
    """Replace the item at position index with the given one.

    Preconditions: 0 <= index < self.length()
    Postconditions: post-self.get_item(index) == item
    """
    pass

def insert(self, index: int, item: object) -> None:
    """Insert item at position index.

    Preconditions: 0 <= index <= self.length() < self.capacity()
    Postconditions: post-self is the sequence
    pre-self.get_item(0), ..., pre-self.get_item(index - 1),
    item, pre-self.get_item(index), ...,
    pre-self.get_item(pre-self.length() - 1)
    """
    pass

def append(self, item: object) -> None:
    """Add item to the end of the sequence.

    Preconditions: self.length() < self.capacity()
    Postconditions: post-self is the sequence
    pre-self.get_item(0), ..., pre-self.get_item(pre-self.
    length() - 1), item
    """
    self.insert(self.length(), item)

def remove(self, index: int) -> None:
```

(continues on next page)

(continued from previous page)

```
"""Remove the item at the given index.

Preconditions: 0 <= index < self.length()
Postconditions: post-self is the sequence
    pre-self.get_item(0), ..., pre-self.get_item(index - 1),
    pre-self.get_item(index + 1), ...,
    pre-self.get_item(pre-self.length() - 1)
"""

pass

def __str__(self) -> str:
    """Return a string representation of the sequence.

    Postconditions: the output uses Python's syntax for lists
"""

    items = []
    for index in range(self.length()):
        items.append(self.get_item(index))
    return str(items)
```

Methods `append` and `__str__` can already be implemented because they don't need to access any instance variable. This saves us from repeatedly implementing them in each subclass because a subclass inherits the methods of its superclass, unless it redefines them. This gives subclasses of `Sequence` the flexibility to either use the above `append` and `__str__` methods or to implement a more efficient version.

Note that the `__str__` method uses the `append` method of the `list` class: methods in different classes can have the same name.

Later optional exercises will ask you to implement the `remove` method.

Exercise 6.3.1

Add to the `Sequence` class a `has` method that implements the membership operation in terms of the other operations.

Hint Answer

6.3.2 Testing data types

Having defined the methods to be implemented, we can write tests for them. Each test takes an empty sequence instance, created by the subclass to be tested, applies operations that modify the sequence and finally uses the operations that inspect sequences to check the result.

In the following, I create a sequence of natural numbers in different ways, by inserting, appending and replacing items, and check the resulting sequence.

The final argument of `check` is always the sequence being tested, so that it's printed if the test fails.

```
[2]: # this code is also in m269_sequence.py
```

```
def test_items(test: str, items: Sequence) -> None:
    """Check that items is the sequence 0, 1, 2, ..., length - 1."""
    for index in range(items.length()):
        check(test + ": n-th item", items.get_item(index), index, items)
    check(
        test + ": length <= capacity", items.length() <= items.capacity(), True, items
    )

def test_init(items: Sequence) -> None:
    """Check that items is the empty sequence."""
    check("init length", items.length(), 0, items)
    test_items("init", items)

def test_append(items: Sequence, length: int) -> None:
    """Check a sequence created with successive appends.

    Preconditions: items is empty; 0 <= length < items.capacity()
    """
    for number in range(length):
        items.append(number)
    test_items("append", items)

def test_insert_start(items: Sequence, length: int) -> None:
    """Check a sequence created by successive inserts at index 0.

    Preconditions: items is empty; 0 <= length <= items.capacity()
    """
    for number in range(length - 1, -1, -1):
        items.insert(0, number)
    test_items("insert at 0", items)

def test_set_item(items: Sequence, length: int) -> None:
    """Check a sequence created by replacing all items.

    Preconditions: items is empty; 0 <= length <= items.capacity()
    """
    for number in range(length): # noqa: B007
        items.append(None)
```

(continues on next page)

(continued from previous page)

```

for number in range(length):
    items.set_item(number, number)
test_items("set item", items)

def test_remove(items: Sequence, length: int) -> None:
    """Check removal at the start, middle and end of a sequence.

    Preconditions: items is empty; 0 <= length <= items.capacity()
    """
    for number in range(length):
        items.append(number)
    if length > 0:
        middle = length // 2
        items.remove(middle)
        check("length decreased", items.length(), length - 1, items)
        check("middle removed", items.has(middle), False, items)
    if items.has(0): # if first number still exists
        pre_length = items.length()
        items.remove(0)
        check("length decreased", items.length(), pre_length - 1, items)
        check("first removed", items.has(0), False, items)
    if items.has(length - 1): # if last number still exists
        pre_length = items.length()
        items.remove(items.length() - 1)
        check("length decreased", items.length(), pre_length - 1, items)
        check("last removed", items.has(length - 1), False, items)
    
```

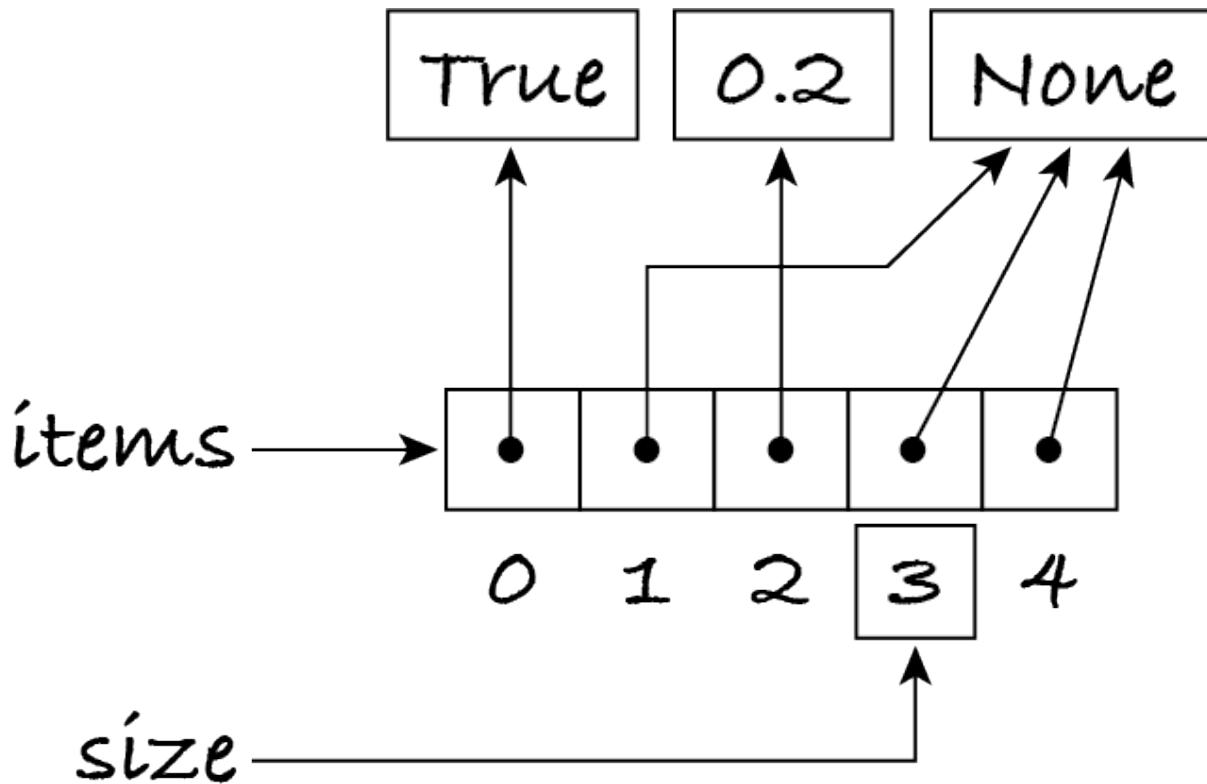
We'll run the tests after implementing each subclass of `Sequence` in the rest of the chapter.

6.4 Bounded sequences

Using a static array, we can implement the sequence ADT for bounded sequences, because static arrays can't grow. The capacity of the sequence is the length of the static array. We must keep track of the sequence's current length, so that we know which positions on the static array are occupied by sequence elements and which ones are available for the sequence to grow. We can't assume that the value `None` represents an available position, because `None` is a potentially valid item of a sequence.

The next figure shows the data structures (a static array for the items and an integer for the size) for a sequence of length three and capacity five, with values `True`, `None` and `0.2`. The static array was initialised with five pointers to the same `None` object, two of which have been replaced. The size of the sequence is also the index of the next available position because indices start at zero. To illustrate this, I write the indices below the array. As I wrote earlier, indices are also a form of reference.

Figure 6.4.1



Most sequence operations are straightforward to implement for bounded sequences, by iterating over one or two static arrays (the input and output sequences). The insert and remove methods require a bit more care. I will implement insertion: the removal algorithm is largely symmetric. Before that, let's take a little detour.

6.4.1 Outlining algorithms

In verbal and written communication with colleagues, algorithms are often presented in a succinct way, just outlining the steps they carry out. This is often sufficient to be able to implement the algorithm. For example, here's how I'd outline the algorithm for the `__str__` method in the abstract `Sequence` class:

The algorithm converts the sequence to a list, which is then converted to a string.
To obtain the list, it starts by creating an empty list. It goes through the items in the sequence, appending each to the list. The list is converted to a string with the `str` function.

Algorithms are commands for the computer to execute, so if you prefer, you can outline them in the imperative tense. Here's the same example:

Convert the sequence to a list and then convert the list to a string. To obtain the list, first create an empty list, then go through the items in the sequence, appending each to the list. Convert the list to a string with the `str` function.

When outlining algorithms, you can assume the reader or listener already knows the problem, i.e. what the inputs and output are and what the algorithm is supposed to accomplish. It's good practice to first give a 'bird's eye view' of the algorithm, describing its various stages or phases,

and then detail each one. This helps the reader or listener see the wood for the trees. An outline that sounds like reading the code aloud is not effective communication: it's too detailed and it overwhelms the recipient because they must keep many details in their head to follow the outline.

For subtle or complicated algorithms, the outline should help the reader or listener understand not just what the algorithm does but also why it works. So do try to explain the rationale of any hairy part of an algorithm.

The outline has to be tailored to the knowledge level of the recipient. The above outlines are Greek to most people, or Latin if they happen to speak Greek. In M269, the recipient is your tutor or a fellow student, so you can assume they have the same knowledge as you.

Like everything else, outlining algorithms requires practice. If you have a study buddy, call them, outline an algorithm in this book and have them critique your outline, e.g. if it's missing important details or if it's too detailed or confusing. If your buddy can follow your outline just by listening to it, without looking at the algorithm in their copy of the book, then it's probably a good one. You can also post and critique outlines in the M269 forums, e.g. of alternative solutions to the book's exercises.

With this in mind, let's tackle the insert operation.

6.4.2 Insertion

As usual, before designing an algorithm, we must think of problem instances. For this operation, the edge cases are lists with length zero and one, and inserting an item at the first, last and after the last index. The latter corresponds to appending the item. The table columns are the variables in the operation definition: *values* is the input/output sequence; *value* and *index* are the item to be inserted and its position. The item to be inserted is not relevant, so I use the same for all test cases. This saves me from writing a very repetitive column for *value*. Remember that we write abstract sequences within parentheses.

Case	Pre-values	index	Post-values
length 0	()	0	('!')
length 1, before	(0)	0	('!', 0)
length 1, after	(0)	1	(0, '!')
length 2, before	(0, 1)	0	('!', 0, 1)
length 2, middle	(0, 1)	1	(0, '!', 1)
length 2, after	(0, 1)	2	(0, 1, '!')

We can assume the bounded sequence has capacity for at least one more item. The algorithm must somehow ‘push’ the items from the given index one position to the right, to then put the item at the position given by the index. Let's take one problem instance to help visualise the process. I choose the fourth test case, inserting at the start of a length 2 sequence, before handling the possibly trickier case of inserting in an empty sequence or inserting at the end.

The sequence is (0, 1). To obtain the end result ('!', 0, 1) I must copy each item to the next position up. I can't start by copying the first item, as it would overwrite the value there and lose it forever: (0, 0). I must start copying from the end:

- (0, 1, 1): copy the last item to the vacant position

- $(0, 0, 1)$: copy the first item to the next position
- $('!', 0, 1)$: put the new item in the input position.

Here's a possible outline of the algorithm:

Go through the items backwards, from the last one to the one at the given index, and copy each item to the next position. Put the given item at the given index.

From this, I can write an algorithm in English. I can't iterate backwards over items, but I can iterate downwards over integers.

1. for each *position* from $|values| - 1$ down to *index*:
 1. let $values[position + 1]$ be $values[position]$
 2. let $values[index]$ be *value*

Step 1.1 copies each item one position up, i.e. to the right.

Before moving on to implementing this algorithm in Python, let's check it works for the other test cases. The penultimate case (inserting in the middle) is similar to inserting at the start. All other cases (first, third and last) append an item. Does the algorithm work for them?

The item is appended when $index = |values|$. In that case, the loop doesn't execute because the start position is larger than the end position. The item is immediately put in its position.

The complexity of this algorithm is as stated in [Section 4.6.1](#): $\Theta(1)$ when appending and $\Theta(|values| - index)$ otherwise.

6.4.3 The `BoundedSequence` class

Translating the insertion algorithm to code requires one more detail: incrementing the size variable, because we're dealing with concrete data structures and not abstract sequences.

Here's an implementation of the abstract class methods for bounded sequences. In Python we write `class A(B)` : to define *A* as a subclass of *B*.

```
[1]: %run -i ../m269_array
%run -i ../m269_sequence

class BoundedSequence(Sequence):
    """A sequence with a fixed capacity."""

    def __init__(self, capacity: int) -> None:
        """Create an empty sequence that can hold 'capacity' items.

        Preconditions: capacity >= 0
        """
        self.items = StaticArray(capacity)
        self.size = 0
```

(continues on next page)

(continued from previous page)

```

def capacity(self) -> int:
    """Return how many items the sequence can hold.

    Postconditions: the output is the value set at creation time
    """
    return self.items.length()

def length(self) -> int:
    """Return the number of items in the sequence.

    Postconditions: 0 <= self.length() <= self.capacity()
    """
    return self.size

def get_item(self, index: int) -> object:
    """Return the item at position index.

    Preconditions: 0 <= index < self.length()
    Postconditions: the output is the n-th item of self, with n-
    =>= index + 1
    """
    return self.items.get_item(index)

def set_item(self, index: int, item: object) -> None:
    """Replace the item at position index with the given one.

    Preconditions: 0 <= index < self.length()
    Postconditions: post-self.get_item(index) == item
    """
    self.items.set_item(index, item)

def insert(self, index: int, item: object) -> None:
    """Insert item at position index.

    Preconditions: 0 <= index <= self.length() < self.capacity()
    Postconditions: post-self is the sequence
    pre-self.get_item(0), ..., pre-self.get_item(index - 1),
    item, pre-self.get_item(index), ...,
    pre-self.get_item(pre-self.length() - 1)
    """
    for position in range(self.size - 1, index, -1):
        self.items.set_item(position + 1, self.items.get_
item(position))
        self.items.set_item(index, item)
        self.size = self.size + 1

```

The following example uses the `append` and `__str__` methods, provided by the abstract superclass.

```
[2]: items = BoundedSequence(5)
print(items)
for item in ("ready", "set", "go!"):
    items.append(item)
    print(items)
print("Can have", items.capacity() - items.length(), "more items.")

[]
['ready']
['ready', 'set']
['ready', 'set', 'go!']
Can have 2 more items.
```

Let's run the tests created in the previous section, for different capacities and lengths. Each test takes an empty bounded sequence of some capacity and grows it to a given length.

```
[3]: %run -i ../m269_test

for capacity in range(4):
    print("Testing capacity", capacity)
    test_init(BoundedSequence(capacity))
    for length in range(capacity + 1):
        print("Testing length", length)
        test_append(BoundedSequence(capacity), length)
        test_insert_start(BoundedSequence(capacity), length)
        test_set_item(BoundedSequence(capacity), length)

Testing capacity 0
Testing length 0
Testing capacity 1
Testing length 0
Testing length 1
Testing capacity 2
Testing length 0
Testing length 1
Testing length 2
insert at 0: n-th item FAILED for [0, None] : None instead of 1
Testing capacity 3
Testing length 0
Testing length 1
Testing length 2
insert at 0: n-th item FAILED for [0, None] : None instead of 1
Testing length 3
insert at 0: n-th item FAILED for [0, None, None] : None instead of 1
insert at 0: n-th item FAILED for [0, None, None] : None instead of 2
```

Oh dear! The test for inserting at the start failed for lists longer than one. In all cases the

resulting sequence is 0 followed by `None`, instead of being 0, 1, 2, ...

Exercise 6.4.1

Explain the error and how to fix it. (By the way, this was a genuine error: I didn't do it on purpose.)

Hint Answer

Exercise 6.4.2

Fix the error in the `insert` method. Rerun the tests.

Answer

Exercise 6.4.3 (optional)

Add a `remove` method to the `BoundedSequence` class above. Run its code cell again, then run these tests:

```
[4]: for capacity in range(4):
    print("Testing capacity", capacity)
    for length in range(capacity + 1):
        print("Testing length", length)
        test_remove(BoundedSequence(capacity), length)
```

Hint

6.5 Dynamic arrays

A **dynamic array** is a data type that adds a `resize` operation to a static array data structure. The operation creates a new static array of the length asked for, copies the items from the current to the new static array, which then becomes the current one. The old static array is garbage collected.

If the length is increased, the extra positions are filled with the default value for the type of items in the static array. If the length is decreased, the excess items are lost. For example, if a dynamic array with integers 3, 5, 7, 9, 11 is resized to a length of three, the new static array has values 3, 5, 7. It's up to dynamic array users to resize them when appropriate for the problem at hand.

A dynamic array has the main advantage of static arrays, i.e. constant-time access and replacement of items, with the flexibility of growing and shrinking as necessary. This is achieved through a succession of static arrays of different lengths.



Info: In some programming languages, a dynamic array is known as a vector.

Shrinking an array can be done in constant time, by simply setting the length to a smaller value to truncate the array, without any copying. This requires memory management support in order to garbage collect part of an existing array, which may not be available on all platforms.

It's thus best to assume the complexity of the resize operation is always linear in the new length, because the operation has to create and initialise a new array of that length and then copy at most that many items to it.

6.5.1 The DynamicArray class

Since a dynamic array can be seen as a kind of static array, I can implement dynamic arrays as a subclass of `StaticArray`, which is defined in `m269_array.py`.

```
[1]: %run -i ../m269_array
```

Subclasses inherit all the operations of their parent class, so I only have to add the resize operation. Methods of subclasses can (and usually have to) access instance variables of their superclass.

The postconditions for the resize operation must state that the new array keeps the same items up to the shorter of the two arrays and the rest, if any, are `None`. I use again the ‘pre- x ’ and ‘post- x ’ notation to refer to the value of variable x before and after the operation.

```
[2]: # this code is also in m269_array.py
```

```
class DynamicArray(StaticArray):
    """An array that can grow and shrink."""

    def resize(self, length: int) -> None:
        """Shorten or extend the array to the new length.

        Preconditions: 0 <= length; length != self.length()
        Postconditions: if pre-self is a_0, a_1, ..., a_n then
            post-self is b_0, b_1, ..., b_m with
            - n == pre-self.length() - 1
            - m == length - 1
            - b_i == a_i for every i from 0 to min(n, m)
            - b_i == None for every i from min(m, n) + 1 to m
        """

        new_array = [None] * length
        for index in range(0, min(length, self.length())):
            new_array[index] = self.items[index]
        self.items = new_array
```

Note that I can call the `length` method on the dynamic array `self` even though the method isn't defined in this class. Every instance of `DynamicArray` is an instance of `StaticArray`, so all methods of the latter also apply to the former.

6.5.2 Tests

The black-box testing of the new method is done as usual: create an array with 0, 1, 2, ..., resize it and check the result with the `length` and `get_item` methods. After resizing, each value is either the same or `None`, as indicated in the postconditions above.

[3]: %run -i ./m269_test

```
def test_resize(old_length: int, new_length: int) -> None:
    """Test resizing a dynamic array from old_length to new_length.

    Preconditions:
    - 0 <= old_length; 0 <= new_length
    - old_length != new_length
    """
    array = DynamicArray(old_length)
    for index in range(old_length):
        array.set_item(index, index)
    array.resize(new_length)
    check("new length", array.length(), new_length, array)
    for index in range(new_length):
        if index < min(old_length, new_length):
            value = index  # old item
        else:
            value = None  # new item
        check("get item", array.get_item(index), value, array)
```

I must test both growing and shrinking. The edge cases are resizing from and to an array of length zero or one, the smallest non-empty array.

[4]:

```
for old in (0, 1, 3, 6):
    for new in (0, 1, 3, 6):
        if old != new:
            print("Resize from", old, "to", new)
            test_resize(old, new)
```

```
Resize from 0 to 1
Resize from 0 to 3
Resize from 0 to 6
Resize from 1 to 0
Resize from 1 to 3
Resize from 1 to 6
Resize from 3 to 0
Resize from 3 to 1
Resize from 3 to 6
Resize from 6 to 0
Resize from 6 to 1
Resize from 6 to 3
```

6.6 Using dynamic arrays

With dynamic arrays, we can now implement the unrestricted sequence ADT: when the capacity is exhausted but an item is about to be inserted, we resize the dynamic array to increase the capacity and then insert the item. It has been shown that the best policy is to set the new capacity in proportion to the current capacity. For example, in the implementation below I set the new capacity to double of the current one.

Let's see why that's a good policy. Resizing a full array, with length and capacity n , takes $\Theta(n)$, in order to copy the n items to the larger static array. Since the new array has capacity $2 \times n$, we can insert n more items without resizing. Spreading the cost of resizing over the subsequent insertions, we see that the overhead per insertion is $\Theta(n) / n = \Theta(1)$. In other words, the run-time of a resize operation is the same as if we added some constant time to each insertion.

Adding a constant time doesn't increase the complexity of an operation, so the complexity of inserting in dynamic arrays is the same as for static arrays: inserting at position i takes $\Theta(n - i)$ and appending ($i = n - 1$) takes $\Theta(1)$.

Python's lists are implemented as dynamic arrays, because this allows lists to grow as necessary, while accessing, replacing and appending items in constant time.



Note: Dynamic arrays are the best data structure for implementing the sequence ADT.

6.6.1 The ArraySequence class

To implement the sequence ADT with dynamic arrays, we need the `Sequence` and `DynamicArray` classes.

```
[1]: %run -i ../m269_array  
%run -i ../m269_sequence
```

The name and docstring of the new subclass reveal which data structure is used, so that users know that indexing, replacing and appending take constant time.

```
[2]: # this code is also in m269_sequence.py  
  
import math  
  
class ArraySequence(Sequence):  
    """A dynamic array implementation of the sequence ADT."""  
  
    def __init__(self) -> None:  
        """Create an empty sequence."""  
        self.items = DynamicArray(1)  
        self.size = 0
```

(continues on next page)

(continued from previous page)

```

def capacity(self) -> float:
    """Return how many items the sequence can hold: infinite."""
    return math.inf # infinite capacity

def length(self) -> int:
    """Return the number of items in the sequence.

    Postconditions: 0 <= self.length() <= self.capacity()
    """
    return self.size

def get_item(self, index: int) -> object:
    """Return the item at position index.

    Preconditions: 0 <= index < self.length()
    Postconditions: the output is the n-th item of self, with n
    =>= index + 1
    """
    return self.items.get_item(index)

def set_item(self, index: int, item: object) -> None:
    """Replace the item at position index with the given one.

    Preconditions: 0 <= index < self.length()
    Postconditions: post-self.get_item(index) == item
    """
    self.items.set_item(index, item

def insert(self, index: int, item: object) -> None:
    """Insert item at position index.

    Preconditions: 0 <= index <= self.length() < self.capacity()
    Postconditions: post-self is the sequence
    pre-self.get_item(0), ..., pre-self.get_item(index - 1),
    item, pre-self.get_item(index), ...,
    pre-self.get_item(pre-self.length() - 1)
    """
    if self.size == self.items.length(): # array full
        self.items.resize(2 * self.size) # double the capacity

    for position in range(self.size - 1, index - 1, -1):
        self.items.set_item(position + 1, self.items.get_
    ↪item(position))
        self.items.set_item(index, item)
    self.size = self.size + 1

```

The following code accesses the instance variables on purpose to show how the internal static array evolves. The array is printed with the `__str__` method inherited from `StaticArray` and the sequence is printed with the `__str__` method inherited from `Sequence`.

```
[3]: sequence = ArraySequence()
print("array", sequence.items, "stores sequence", sequence)
for value in range(0, 5):
    sequence.append(value)
    print("array", sequence.items, "stores sequence", sequence)

array [None] stores sequence []
array [0] stores sequence [0]
array [0, 1] stores sequence [0, 1]
array [0, 1, 2, None] stores sequence [0, 1, 2]
array [0, 1, 2, 3] stores sequence [0, 1, 2, 3]
array [0, 1, 2, 3, 4, None, None, None] stores sequence [0, 1, 2, 3, ↵4]
```

As we can see, the length of the static array doubles step-wise from 1 to 8, and the unused positions have value `None`.

Finally, let's test each method.

```
[4]: %run -i ../m269_test

test_init(ArraySequence())
for length in range(10):
    print("Testing length", length)
    test_append(ArraySequence(), length)
    test_insert_start(ArraySequence(), length)
    test_set_item(ArraySequence(), length)

Testing length 0
Testing length 1
Testing length 2
Testing length 3
Testing length 4
Testing length 5
Testing length 6
Testing length 7
Testing length 8
Testing length 9
```

Exercise 6.6.1 (optional)

Add a `remove` method to the `ArraySequence` class, then run these tests:

```
[5]: %run -i ../m269_test
```

(continues on next page)

(continued from previous page)

```
for length in range(5):
    print("Testing length", length)
    test_remove(ArraySequence(), length)
```

After the tests pass, add code to your `remove` method to shrink the array if the sequence, after the item was removed, is much shorter than the array. It's up to you what 'much shorter' means.

You must shrink the capacity to a value that is proportional to the current one, in order to keep the overhead per removal constant. In addition, you must leave some spare capacity after shrinking: otherwise the next `insert` would make the array grow again. Having to grow a dynamic array right after shrinking it wouldn't be efficient.

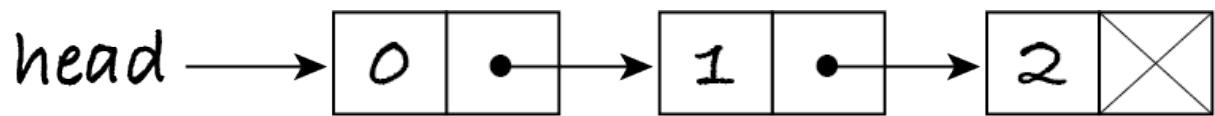
Hint

6.7 Linked lists

Arrays keep all data (or pointers to data) in contiguous memory to support constant-time access. Another approach to implementing sequences scatters the data over memory. There's a reason and a method to this apparent madness.

A **linked list** is a chain of **nodes**, each with an element of the sequence (or pointer to it) and a pointer to the next node. The first node is called the **head** of the linked list. The last node has a **null pointer**, as there's no next node. The next figure shows a linked list with integers 0, 1 and 2. The X represents the null pointer and marks the end of the linked list.

Figure 6.7.1



The `head` variable refers to the first node. If `head` is the null pointer, then the list is empty.

6.7.1 Traversing a linked list

Traversing a collection means to go through the collection's items, one by one. This is typically done with iteration: a for- or while-loop. (Chapter 12 will show how to traverse collections without iteration.)

Assuming each node is an object with two instance variables `item` and `next`, then an algorithm to traverse a linked list and process each item is:

1. let `current` be `head`
2. while `current` isn't the null pointer:
 1. process `current.item`
 2. let `current` be `current.next`

Step 2.1 does whatever is needed for the problem at hand and step 2.2 updates the reference to now refer to the next node. If the linked list is empty, the step 2 condition is false and the loop doesn't execute.

This algorithm can be adapted to access the item at a given index of the sequence by replacing the while-loop with a for-loop. This means that accessing an item takes linear time with linked lists, more specifically $\Theta(i)$ to get the item at index i , whereas with arrays it takes constant time.

6.7.2 Inserting an item

The algorithm to insert an item *value* at position *index* is more subtle. Here's an outline of it:

First iterate over the linked list to obtain references to the nodes that will be before and after the new item. The 'before' node is at position $index - 1$; the 'after' node is at position $index$. Create a new node with the item to be inserted. Make the 'before' node point to the new node, and the new node point to the 'after' node. In this way the new node is now at position $index$ and the 'after' node (and the rest of the list) has been 'pushed' one position up.

To illustrate the algorithm, let's insert integer 3 in the sequence (0, 1, 2) at index 2, i.e. the resulting sequence should be (0, 1, 3, 2).

First we obtain references to the node at positions $index - 1$ and $index$:

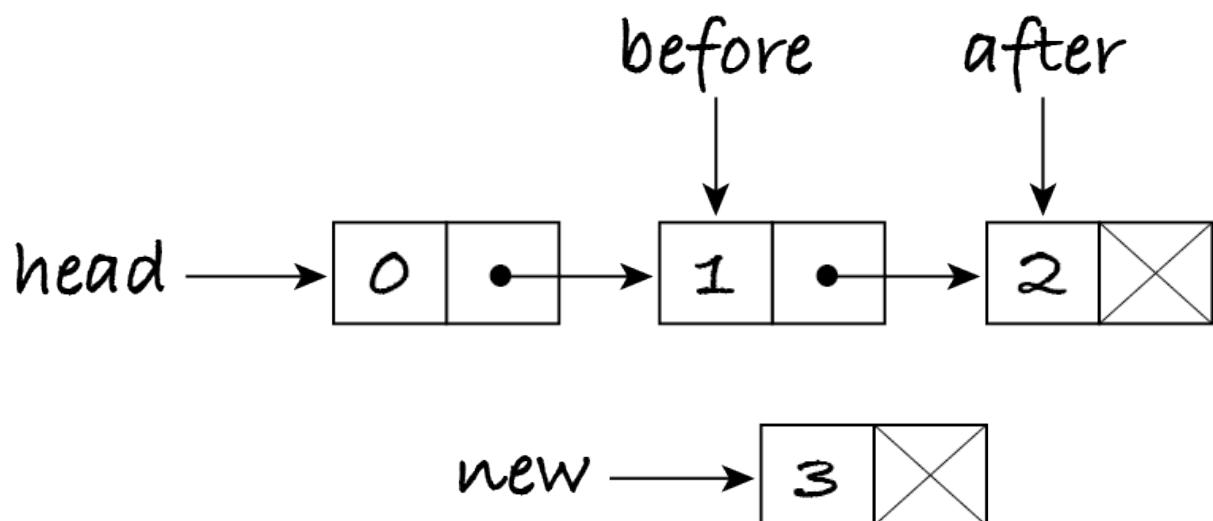
1. let *before* be *head*
2. repeat $index - 1$ times:
 1. let *before* be *before.next*
 3. let *after* be *before.next*

Next we create a new node with the item to be inserted but without a next node:

4. let *new* be a node with *item* = *value* and *next* = null pointer

For our example the situation at this stage is:

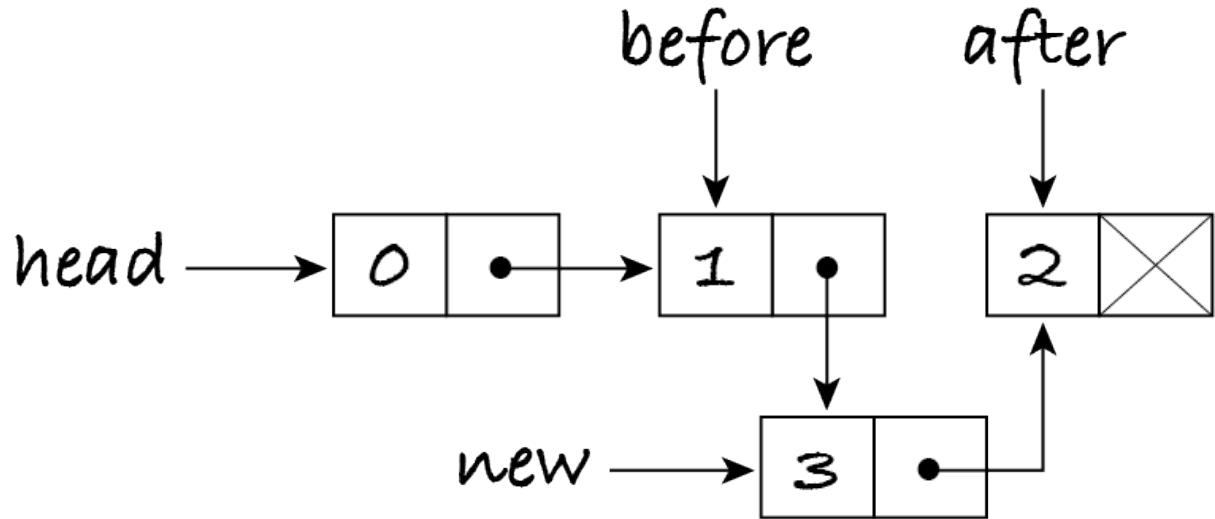
Figure 6.7.2



Finally we change the pointers to put the new node between *before* and *after*.

5. let *before.next* be *new*
6. let *new.next* be *after*

Figure 6.7.3



Once the algorithm knows where to insert the new item, the insertion itself takes constant time: no copying of values takes place.

Let's check the algorithm with our test table.

Case	Pre-values	<i>index</i>	Post-values
length 0	()	0	('!')
length 1, before	(0)	0	('!', 0)
length 1, after	(0)	1	(0, '!')
length 2, before	(0, 1)	0	('!', 0, 1)
length 2, middle	(0, 1)	1	(0, '!', 1)
length 2, after	(0, 1)	2	(0, 1, '!')

For the first test case (empty sequence), the *head* variable is the null pointer and so is *before* after step 1. Variable *index* has value zero, so the loop is skipped, because it can't be executed minus one times. Step 3 tries to access instance variable *next* but *before* is not pointing to a node. This kind of error is called **null pointer dereference**: we're trying to dereference (i.e. access the object pointed by) *before*, but *before* is not a valid pointer.

A quick fix to the algorithm is to move step 4 (the creation of the new node) to the beginning and then handle empty and non-empty sequences separately.

1. let *new* be a node with *item = value* and *next = null pointer*
2. if *head* is the null pointer:
 1. let *head* be *new*
 3. otherwise:

1. let *before* be *head*
2. repeat *index* – 1 times:
 1. let *before* be *before.next*
 3. let *after* be *before.next*
 4. let *before.next* be *new*
 5. let *new.next* be *after*

Let's move on to the second test: inserting the item at the start of a sequence of length one. Is the algorithm correct for this case?

Alas, it isn't. If an item is inserted at the start, we must update the *head* variable to refer to the new node, but the algorithm never does so.

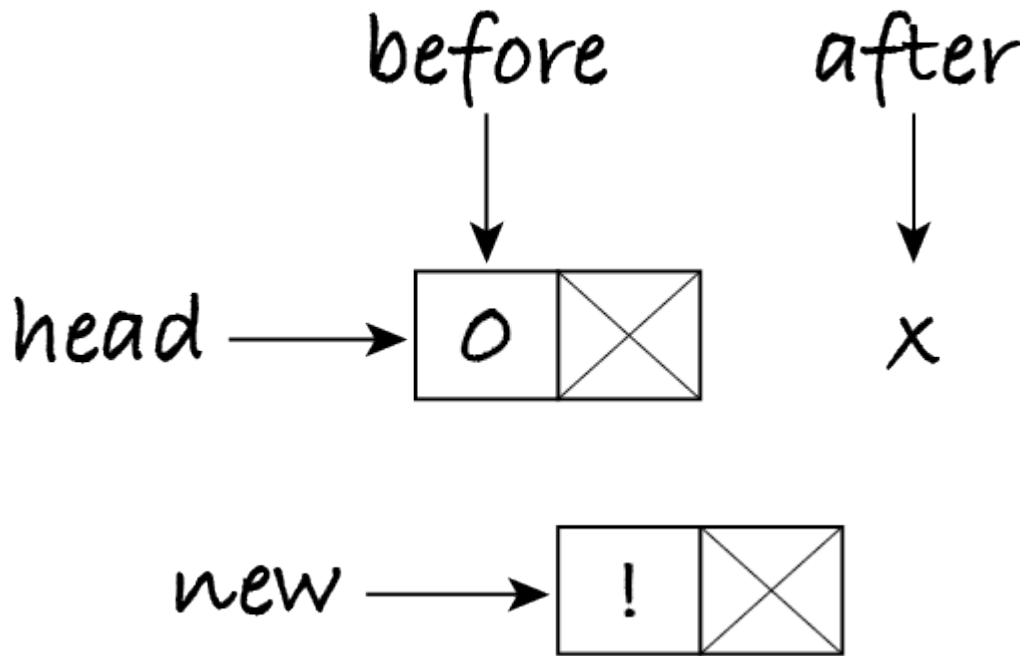
We can fix the algorithm by treating this edge case separately. The current head node becomes the node after the new node, which in turn becomes the head node.

1. let *new* be a node with *item* = *value* and *next* = null pointer
2. if *head* is the null pointer:
 1. let *head* be *new*
3. otherwise if *index* = 0:
 1. let *after* be *head*
 2. let *head* be *new*
 3. let *new.next* be *after*
4. otherwise:
 1. let *before* be *head*
 2. repeat *index* – 1 times:
 1. let *before* be *before.next*
 3. let *after* be *before.next*
 4. let *before.next* be *new*
 5. let *new.next* be *after*

Let's move on to the third test, which inserts the item at index one of a sequence of length one, i.e. it appends the item. Is the algorithm correct for this case?

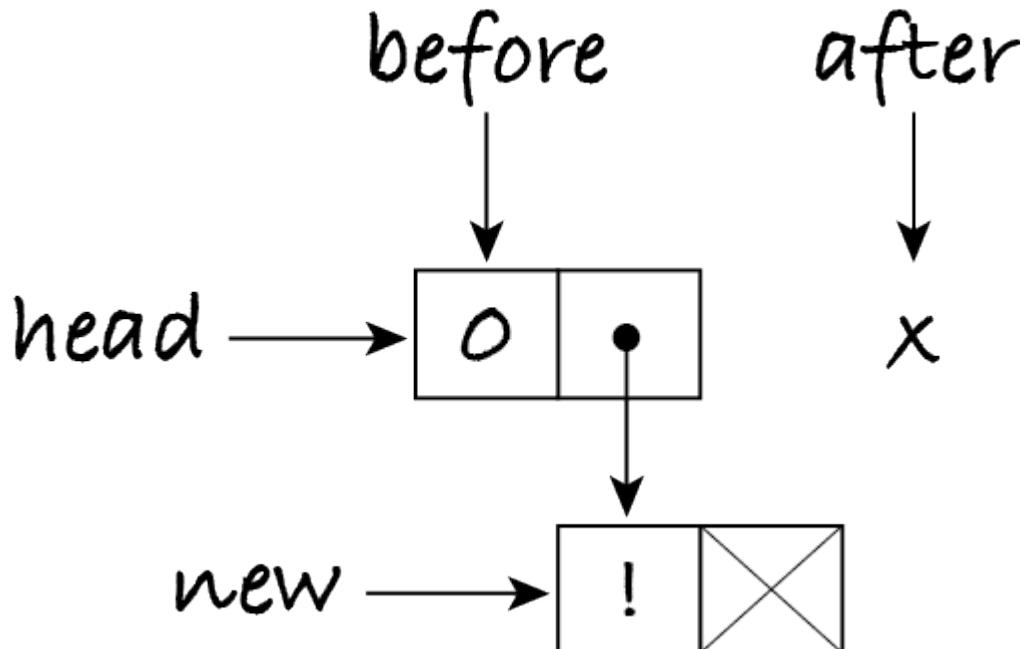
The sequence not being empty and the index not being zero, the algorithm executes step 1 and then step 4.1, making *before* refer to the first and only node in the linked list. The loop is repeated zero times, because *index* = 1. Step 4.3 sets *after* to be the null pointer. The situation is:

Figure 6.7.4



Step 4.4 links the ‘before’ node to the new node. Step 4.5 has no practical effect, because the new node’s *next* variable is already the null pointer. The final situation is as follows. The algorithm correctly appends items.

Figure 6.7.5



The algorithm works when there’s no node after the new node. This makes me realise that the part that handles insertions at the start (steps 3 to 3.3) also works for the empty list, when *head* and *after* are the null pointer. I can eliminate steps 2 and 2.1. I actually don’t need variable *after* and can reduce steps 3.1 to 3.3 and 4.3 to 4.5 to just two steps each. Here’s my final algorithm:

1. let *new* be a node with *item* = *value* and *next* = null pointer
2. if *index* = 0:

1. let *new.next* be *head*
2. let *head* be *new*
3. otherwise:
 1. let *before* be *head*
 2. repeat *index* – 1 times:
 1. let *before* be *before.next*
 3. let *new.next* be *before.next*
 4. let *before.next* be *new*

Exercise 6.7.1

Outline an algorithm to remove the item at a given position *index*.

Hint Answer

6.7.3 The `LinkedSequence` class

Python doesn't allow us to manipulate pointers (memory addresses) directly: we can only refer to objects via variables. The most natural choice is to represent a null pointer as a variable with value `None`, but any other object that hasn't got instance variables named 'next' and 'item' can be used: in this way a null pointer dereference becomes an attribute error in Python.

```
[1]: node = None
node.next
-----
→-----
AttributeError                                     Traceback (most recent...
→call last)
Cell In[1], line 2
    1 node = None
--> 2 node.next

AttributeError: 'NoneType' object has no attribute 'next'
```

The node objects are just data: they don't have operations. Writing a class with two instance variables and four methods to access and modify the variables is overkill. Since nodes are only meaningful in the context of linked lists, I define a `Node` class *within* the `LinkedSequence` class, and let the methods of the latter access the instance variables of nodes. This saves us from writing four trivial methods while keeping nodes hidden from users of sequences. Since `Node` is within `LinkedSequence`, the constructor must be called by its full name: `LinkedSequence.Node(item)`.

Here's the class. Most methods of `Sequence` are implemented by traversing the linked list as explained above.

```
[2]: %run -i ./m269_sequence

import math

class LinkedSequence(Sequence):
    """A linked list implementation of the sequence ADT."""

    class Node:
        """A node in a linked list."""

        def __init__(self, item: object) -> None:
            """Initialise the node with the given item."""
            self.item = item
            self.next = None

    def __init__(self) -> None:
        """Initialise the sequence to be empty."""
        self.head = None

    def capacity(self) -> float:
        """Return how many items the sequence can hold: infinite."""
        return math.inf # infinite capacity

    def length(self) -> int:
        """Return the number of items in the sequence.

        Postconditions: 0 <= self.length() <= self.capacity()
        """
        size = 0
        current = self.head
        while current != None:
            size = size + 1
            current = current.next
        return size

    def get_item(self, index: int) -> object:
        """Return the item at position index.

        Preconditions: 0 <= index < self.length()
        Postconditions: the output is the n-th item of self, with n <= index + 1
        """
        current = self.head
        for times in range(index): # noqa: B007
            current = current.next
        return current.item
```

(continues on next page)

(continued from previous page)

```
def set_item(self, index: int, item: object) -> None:
    """Replace the item at position index with the given one.

    Preconditions: 0 <= index < self.length()
    Postconditions: post-self.get_item(index) == item
    """
    current = self.head
    for times in range(index):  # noqa: B007
        current = current.next
    current.item = item

def insert(self, index: int, item: object) -> None:
    """Insert item at position index.

    Preconditions: 0 <= index <= self.length() < self.capacity()
    Postconditions: post-self is the sequence
                    pre-self.get_item(0), ..., pre-self.get_item(index - 1),
                    item, pre-self.get_item(index), ...,
                    pre-self.get_item(pre-self.length() - 1)
    """
    new = LinkedSequence.Node(item)
    if index == 0:
        new.next = self.head
        self.head = new
    else:
        before = self.head
        for times in range(index - 1):  # noqa: B007
            before = before.next
        new.next = before.next
        before.next = new
```

And again, let's test the operations.

```
[3]: %run -i ../m269_test

test_init(LinkedSequence())
for length in range(10):
    print("Testing length", length)
    test_append(LinkedSequence(), length)
    test_insert_start(LinkedSequence(), length)
    test_set_item(LinkedSequence(), length)

Testing length 0
Testing length 1
Testing length 2
Testing length 3
```

(continues on next page)

(continued from previous page)

```
Testing length 4
Testing length 5
Testing length 6
Testing length 7
Testing length 8
Testing length 9
```

Exercise 6.7.2 (optional)

Add a `remove` method to the `LinkedSequence` class and test it with this code:

```
[4]: for length in range(5):
    print("Testing length", length)
    test_remove(LinkedSequence(), length)
```

6.7.4 Linked list v. array

The sequence ADT can be implemented with dynamic arrays and with linked lists. The choice depends on which operations we require to be most efficient. Here's a table of the complexities for some operations on sequence s and index i .

Sequence operation	Dynamic array	Linked list
get item at i	$\Theta(1)$	$\Theta(i)$
replace item at i	$\Theta(1)$	$\Theta(i)$
insert at $i = 0$	$\Theta(s)$	$\Theta(1)$
insert at $i = s $ (append)	$\Theta(1)$	$\Theta(s)$
insert elsewhere	$\Theta(s - i)$	$\Theta(i)$

The main advantage of arrays over linked lists is the constant-time access to items, whereas linked lists have to be traversed. Doing operations at the start of a list can be efficient, and we'll take advantage of that in the next chapter, because linked lists don't require copying items when inserting or removing one. Linked lists are never resized. They require more memory than arrays (one pointer per item), but dynamic arrays also waste memory when the size is lower than the capacity.

Some operations on linked lists can become more efficient with extra data. The implementation above computes the length in linear time, by counting items while iterating over the linked list. It's also possible to obtain the length in constant time, by adding an instance variable that is initially zero and is incremented (or decremented) when an item is inserted (or removed), as done with dynamic arrays. This is an example of a **space–time tradeoff**: we are willing to increase the memory usage of a linked list object to reduce the run-time of an operation.



Note: Each data structure makes some operations more efficient than others. The best data structure for the problem at hand is the one that favours the operations we need more frequently.

Exercise 6.7.3

How could you make the append operation take constant time on linked lists?

Hint Answer

6.8 Summary

6.8.1 Classes

A **class** defines a data type in Python. A value of a type defined by a class C is said to be an **object** and in particular, an **instance of class C** . The instance variables hold the data for each instance of the class. A method is an operation provided by a class. In Python, the first argument of each method for class C is an instance of C . Methods often modify the class instance to which they are applied and in such cases do not implement a mathematical function.

The methods form the class's **interface**. A class may change its instance variables without changing its interface. A class's **attributes** are its instance variables and methods.

To define a class C , write:

```
[1]: class C:  
    """What class C represents."""  
  
    def __init__(self) -> None:  
        """Initialise a new instance of C."""  
        # create the instance variables with self.x = value  
  
    # define further methods
```

If all instances of class A are also instances of class B , then A is a **subclass** of B and B a **superclass** of A . Class A inherits the methods of B . In Python, write `class A(B)` : to define a subclass of B .

An **abstract class** isn't meant to be instantiated. It usually doesn't have an `__init__` method nor does it implement most methods. It can be used to define an ADT.

In Python, if a class `Auxiliary` is defined within a class `Main`, then it must be referred to by its full name: `Main.Auxiliary`.

White-box tests access the instance variables while black-box tests only use a class's interface to test the behaviour of each method. Running the same tests after a change is called regression testing. Black-box testing facilitates regression testing, as changes to instance variables and method bodies don't require changing the black-box tests.

6.8.2 Data structures

A data type implements an ADT with a **data structure**, a particular way of organising data. A linear data structure organises data sequentially.

A **static array** is a data structure that stores a fixed-length sequence contiguously in memory, with the same number of bytes per item. This allows any item to be accessed and replaced in constant time.

In Python, a static array can be emulated by creating a list with `[initial_value] * length` and never applying operations that change its length.

A sequence can be stored in a static array of references to the actual items. A **reference** is an object that refers another one. A **pointer** is the memory address of the referred object.

Garbage collection is the process of freeing the memory occupied by objects that are no longer used because no object refers to them.

A **bounded** sequence has a fixed **capacity**, the maximum number of items it can hold, set at creation time. The sequence is full when its length and capacity are equal. A bounded sequence can be implemented with a static array and a variable to keep track of the sequence's length.

A **dynamic array** is a succession of static arrays to give the illusion that the length of the dynamic array shrinks and grows. Resizing the dynamic array involves copying the items from the current static array to a new static array with the new length, and therefore takes time linear in the new length. Python lists are implemented with dynamic arrays.

A **linked list** is a chain of nodes, each holding an item and pointing to the next node. The last node has a **null pointer** to indicate there's no further node. A null pointer dereference occurs when trying to access the object referenced by a pointer without first checking that it's the null pointer. The first node is called the **head** of the list. Linked lists don't require a resize operation nor shifting items when inserting or removing one.

6.8.3 Complexity

Implementing operations may require making **space–time tradeoffs** to reduce the run-time by using extra memory.

The complexity of an operation depends on the underlying data structure. In the following table, s is a sequence and i is the index of the item to access, replace, insert or remove. Contrary to the table in [Section 6.7.4](#), this one accounts for the space–time tradeoffs mentioned in that section.

Sequence operation	Dynamic array	Linked list
length	$\Theta(1)$	$\Theta(1)$
indexing, replace	$\Theta(1)$	$\Theta(i)$
insert, remove	$\Theta(s - i)$	$\Theta(i)$
append	$\Theta(1)$	$\Theta(1)$

6.8.4 Python

The Python statement `pass` does nothing. It can be used to indicate that abstract class methods do nothing: they have to be implemented in a subclass.

Module `math` defines constant `inf` for positive infinity. It can be used to represent the capacity of an unbounded sequence. Negative infinity is represented as `-inf`.

CHAPTER 7

STACKS AND QUEUES

The sequence ADT allows any item to be accessed, replaced or removed, and to insert new items anywhere. In some situations we don't need such flexibility. This chapter introduces three restricted sequence ADTs with examples of their use: stacks, queues and priority queues. They only allow us to access or remove one particular item: the most recently added item (stacks), the least recently added item (queues) or the highest-priority item (priority queues). The chapter also shows how to implement these ADTs.

This chapter supports the following learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn three ADTs (stacks, queues, priority queues) and how to implement them with arrays and linked lists.

This chapter includes further examples to support these learning outcomes:

- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders.
- Analyse the complexity of algorithms to support software design choices.
- Write readable, tested, documented and efficient Python code.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

7.1 Stacks

A pile of boxes is a stack. Only the topmost box can be easily accessed. To access any other box, one usually must remove boxes from the top, one by one. To re-form the pile, one must put each box back on top of the previous boxes.

Computationally, a **stack** is a restricted sequence in which new items are added to and removed from only one end of the sequence. A stack can be seen as a sequence ordered by ‘age’. The ‘oldest’ item, the one added first, is at the bottom of the stack. The ‘youngest’ item, the one

added last, is at the top. A stack is a **last-in, first-out (LIFO)** sequence: the last item to be added is the first to be removed.



Info: Some texts also use FILO (first-in, last-out) as a synonym for LIFO.

7.1.1 The stack ADT

The stack ADT has five operations, where s is some stack:

Operation	Effect	Algorithm in English
new	create a new empty stack	let s be an empty stack
size	obtain the size of s	$ s $
push	put an item on top of s	push <i>item</i> on s
pop	remove the top item, if s isn't empty	pop s
peek	access the top item, if s isn't empty	top of s

A common alternative pop operation also returns the removed item. This makes the peek operation redundant: the top item can be inspected by popping and pushing it.

A stack is a sequence of objects and so we can use sequence ADT operations in defining the stack ADT. The push, pop and peek operations are simply restricted versions of the insert, remove and indexing operations of the sequence ADT.

ADT: stack, a sequence of objects

Function: new

Inputs: none

Preconditions: true

Output: $items$, a stack

Postconditions: $items = ()$

Function: size

Inputs: $items$, a stack

Preconditions: true

Output: $count$, an integer

Postconditions: $count = |items|$

Exercise 7.1.1

Here again is the definition of the insert operation of the sequence ADT from [Section 4.6](#). Modify it for the push operation for a stack.

Operation: insert

Inputs/Outputs: $values$, a sequence

Inputs: $index$, an integer; $value$, an object

Preconditions: $0 \leq index \leq |values|$

Postconditions: post-values = (pre-values[0], ..., pre-values[index - 1], value, pre-values[index], ..., pre-values[|pre-values| - 1])

Hint Answer

Exercise 7.1.2

Modify the definitions of the general remove and indexing operations to become the definitions of the pop and peek operations.

Operation: remove

Inputs/Outputs: $values$, a sequence

Inputs: $index$, an integer

Preconditions: $0 \leq index < |values|$

Postconditions: post-values = (pre-values[0], ..., pre-values[index - 1], pre-values[index + 1], ..., pre-values[|pre-values| - 1])

Function: indexing

Inputs: $values$, a sequence; $index$, an integer

Preconditions: $0 \leq index < |values|$

Output: $value$, an object

Postconditions: $value$ is the n -th item of $values$, with $n = index + 1$

Hint Answer

7.1.2 Implementing with an array

Python doesn't provide a stack data type because stacks can be easily simulated with lists, as long as we only access, add and remove the last item. Here's a list being used as a stack:

```
[1]: numbers = [] # stack
for number in range(3):
    print("push", number)
    numbers.append(number)
while len(numbers) > 0:
    print("pop", numbers[-1])
    numbers.pop(-1)

push 0
push 1
push 2
pop 2
pop 1
pop 0
```

This example clearly shows the last-in, first-out behaviour of stacks: the items are popped in the opposite order they were pushed.

Since Python's `pop` method also returns the removed item, it would be daft not to make use of it when convenient. The last two lines of the cell above can be rewritten as a single line:
`print('pop', numbers.pop(-1)).`

Lists are implemented with dynamic arrays, which means that all stack operations have constant-time complexity as they don't involve any shifting of items.

Using list operations makes it harder to spot that a list is being used as a stack, and this may lead to errors when the code needs to be changed. We therefore implement the ADT with our own class, which simply uses a Python list and calls the corresponding operations.

```
[2]: # this code is also in m269_stack.py

class Stack:
    """A last-in, first-out sequence of objects, implemented with a
    →Python list."""

    def __init__(self) -> None:
        """Initialise the stack to be empty."""
        self.items = []

    def size(self) -> int:
        """Return the number of items in the stack."""
        return len(self.items)

    def peek(self) -> object:
        """Return the top item in the stack.

        Preconditions: self.size() > 0
        """
        return self.items[-1]

    def pop(self) -> object:
        """Remove and return the top item from the stack.

        Preconditions: self.size() > 0
        """
        return self.items.pop(-1)

    def push(self, item: object) -> None:
        """Put the given item on top of the stack.

        Postconditions: post-self.peek() == item
        """
        self.items.append(item)
```

Let's test the class with the same example as before:

```
[3]: numbers = Stack()
for number in range(3):
    print("push", number)
    numbers.push(number)
while numbers.size() > 0:
    print("pop", numbers.peek())
    numbers.pop()

push 0
push 1
push 2
pop 2
pop 1
pop 0
```



Info: Class `Stack` in package `java.util` implements the stack ADT in Java. It's a subclass of `java.util.Vector`, a dynamic array data type.

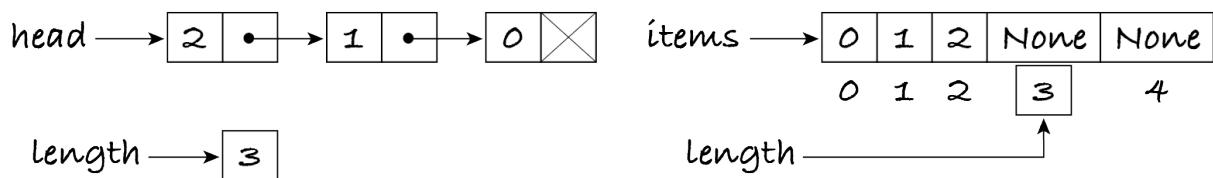
We can implement a bounded stack with a static array, like we did for a *bounded sequence*, by keeping track of the current size of the stack.

7.1.3 Implementing with a linked list

The stack ADT can also be implemented with the linked list data structure. With an array, the top item is the *last* one, at the highest index, so that the push and pop operations don't have to shift items. With a linked list, the top item is the *first* one, in the head node, so that the push and pop operations don't have to traverse the linked list.

Here's the same stack, obtained by pushing 0, 1 and 2 in this order, stored in a static array with capacity 5, and in a linked list. In both cases, an additional instance variable keeps track of the length.

Figure 7.1.1



The implementation with a linked list is a special case of implementing sequence operations with a linked list, e.g. the push operation corresponds to *inserting at index 0*. Each stack operation takes constant time, as it involves a few assignments to update pointers and an integer operation to update the length.

```
[4]: # this code is also in m269_stack.py
```

```
class LinkedListStack:  
    """A last-in, first-out sequence of objects, implemented with a  
    linked list."""  
  
    class Node:  
        """A node in a linked list."""  
  
        def __init__(self, item: object) -> None:  
            """Initialise the node with the given item."""  
            self.item = item  
            self.next = None  
  
        def __init__(self) -> None:  
            """Initialise the stack to be empty."""  
            self.head = None  
            self.length = 0  
  
        def size(self) -> int:  
            """Return the number of items in the stack."""  
            return self.length  
  
        def peek(self) -> object:  
            """Return the top item in the stack.  
  
            Preconditions: self.size() > 0  
            """  
            return self.head.item  
  
        def pop(self) -> object:  
            """Remove and return the top item from the stack.  
  
            Preconditions: self.length() > 0  
            """  
            item = self.head.item  
            self.head = self.head.next  
            self.length = self.length - 1  
            return item  
  
        def push(self, item: object) -> None:  
            """Put the given item on top of the stack.  
  
            Postconditions: post-self.peek() == item  
            """  
            node = LinkedListStack.Node(item)
```

(continues on next page)

(continued from previous page)

```

        node.next = self.head
        self.head = node
        self.length = self.length + 1
    
```

And once more the same test:

```

[5]: numbers = LinkedListStack()
for number in range(3):
    print("push", number)
    numbers.push(number)
while numbers.size() > 0:
    print("pop", numbers.peek())
    numbers.pop()

push 0
push 1
push 2
pop 2
pop 1
pop 0
    
```

Exercise 7.1.3

Is it better to implement stacks with dynamic arrays or linked lists?

Hint Answer

7.2 Using stacks

Let's look at two classic applications of stacks.

7.2.1 Balanced brackets

Python expressions use round and square brackets. The brackets must be balanced: opening and closing brackets must match in pairs, and the brackets enclosed within a pair of matching brackets must also be balanced. If the brackets are not balanced, a syntax error occurs.

```

[1]: ('hello'[1:3 + 'h')] * 3      # brackets within () aren't balanced
      Cell In[1], line 1
      ('hello'[1:3 + 'h')] * 3      # brackets within () aren't balanced
                                         ^
SyntaxError: closing parenthesis ')' does not match opening parenthesis '['
    
```

Given a string, containing Python code or some other text, we want to know if the brackets within that string are balanced. That's a decision problem.

Function: is balanced?

Inputs: *text*, a string

Preconditions: each bracket in *text* is one of (,), [,]

Output: *balanced*, a Boolean

Postconditions: *balanced* is true if and only if

- all opening and closing brackets in *text* match (have the same shape) in pairs
- for each pair of matching brackets, the substring they delimit is balanced

For the test table we need balanced and unbalanced brackets, nested and not. We must think of the various ways in which brackets are unbalanced. The empty string, the absence of brackets, and a string only with brackets are edge cases.

```
[2]: from algoesup import check_tests

is_balanced_tests = [
    # case,           text,                      balanced
    ['no text',      '',                         True],
    ['no brackets',  'brackets are like Russian dolls', True],
    ['matched',      '(3 + 4)',                  True],
    ['mismatched',  '(3 + 4]',                 False],
    ['not opened',   '3 + 4)',                  False],
    ['not closed',   '(3 + 4',                   False],
    ['wrong order',  'close ) before open (',  False],
    ['nested',       '([([])])',                True],
    ['nested pair',  'items[(i - 1):(i + 1)]', True]
]

check_tests(is_balanced_tests, [str, bool])
```

OK: the test table passed the automatic checks.

I told you at the start of this section that this problem requires a stack. If I hadn't let the cat out of the bag, one way to spot a stack is to look for a LIFO relationship. In a balanced string, the *first* closing bracket must match the *last* (most recent) opening bracket, or vice versa the *first* opening bracket must match the *last* closing bracket.

By the way, this problem is a particular case of the more general problem of matching delimiters that can be nested. Any language that has them, like the opening and closing tags in HTML and XML, requires a stack to be processed.



Note: When processing a nested structure, consider a stack.

The algorithm must check if each closing bracket matches the most recent opening bracket. For that, we must keep a stack of the brackets opened so far. As we go through the string, we push opening brackets on the stack, and we pop them as we go through the matching closing brackets.

Let's work through the last test in the table, as it has both kinds of brackets and they're nested. Characters that aren't brackets are skipped.

Stack	Remaining string	Action
['(i - 1):(i + 1)]'	opening bracket: push
[('):(i + 1)]'	closing bracket matches top item: pop
[('(i + 1)]'	opening bracket: push
[()	']'	closing bracket matches top item: pop
[]	']'	closing bracket matches top item: pop

We reached the end of the string. Since the stack is empty, i.e. there are no open but yet unclosed brackets, the brackets must be balanced.

The other tests with balanced brackets are checked in a similar way, so let's consider the tests with unbalanced brackets. As the algorithm goes through the string, how can it spot that the fourth test, '(3 + 4]', isn't balanced?

When it reaches the closing square bracket, the stack has the opening round bracket. The brackets don't match, so the algorithm can immediately stop with *balanced* = false.

Let's consider now the 'not opened' case: '3 + 4]'. How can the algorithm spot the brackets are unbalanced?

When it reaches the closing square bracket, the stack is empty. There's no opening bracket the closing bracket can match, so the algorithm can immediately stop with a false output.

The 'wrong order' case, 'close) before open (', is processed in the same way, so let's look at the final 'not closed' case: '(3 + 4'. How can the algorithm detect it's unbalanced?

Upon reaching the end of the string, the stack isn't empty. Some brackets were opened but not closed, which means they're unbalanced.

Now that we have thought through the test cases, we can outline the algorithm. Here's one possible way, using the imperative tense and describing some of the rationale.

Create an empty stack for the brackets opened so far and not yet matched. Iterate over the string and process each character. If it's an opening bracket, push it on the stack. If it's a closing bracket and the stack is empty or the top item isn't a matching opening bracket, stop: the brackets aren't balanced. Otherwise, the brackets match, so pop the opening bracket from the stack. If the end of the string is reached, the brackets are balanced if and only if the stack is empty, i.e. no open brackets remain to be closed.

Here's the step-by-step algorithm.

1. let *opened* be an empty stack
2. for each *character* in *text*:

1. if $character = '('$ or $character = '['$:
 1. push $character$ on $opened$
2. otherwise if $character = ')'$:
 1. if $|opened| > 0$ and top of $opened = '('$:
 1. pop $opened$
 2. otherwise:
 1. let $balanced$ be false
 2. stop
3. otherwise if $character = ']'$:
 1. if $|opened| > 0$ and top of $opened = '['$:
 1. pop $opened$
 2. otherwise:
 1. let $balanced$ be false
 2. stop
3. let $balanced$ be $|opened| = 0$

Exercise 7.2.1

Give a best- and a worst-case scenario of the algorithm and their corresponding complexities.

Hint Answer

Exercise 7.2.2

Translate the algorithm to Python code.

```
[3]: %run -i ../m269_stack

from algoesup import test

def is_balanced(text: str) -> bool:
    """Check if all brackets in text are balanced.

    Preconditions: each bracket in 'text' is one of (, ), [,
    """
    pass

test(is_balanced, is_balanced_tests)
```

Answer

Exercise 7.2.3

Assuming only round brackets (parentheses) can occur in the input string, outline a simpler linear-time algorithm that checks if they're balanced.

Hint Answer

7.2.2 Postfix expressions

An expression like $3 - 4$ is in **infix** notation: the operator is between the operands. An expression like $3 4 -$ is in **postfix** notation: the operator comes after the operands.

Postfix expressions don't require parentheses because they show explicitly in which order the operators are applied. Some examples:

Infix	Postfix
$3 - 4 \times 5$	$3 4 5 \times -$
$(3 - 4) \times 5$	$3 4 - 5 \times$
$(3 - 4) \times (5 - 6)$	$3 4 - 5 6 - \times$
$3 - 4 \times (5 - 6)$	$3 4 5 6 - \times -$



Info: Postfix notation is also known as reverse Polish notation.

In the following, postfix expressions only contain natural numbers and the subtraction and multiplication operators.

Exercise 7.2.4

Postfix expressions can be evaluated in a single pass from left to right, using a stack. Explain why.

Hint Answer

Exercise 7.2.5

Outline an algorithm that evaluates a postfix expression, given as a non-empty sequence of characters ‘-’ and ‘*’ and natural numbers. You can assume the sequence represents a valid postfix expression, e.g. it doesn't start with an operator.

Hint Answer

Exercise 7.2.6

Implement the algorithm. You don't need to add tests.

```
[4]: %run -i ../m269_stack

from algoesup import test

def evaluate(expression: list) -> int:
    """Return the value of the given postfix expression.

    Preconditions:
    - each item in the input list is a natural number, '-' or '*'
    - the input represents a valid non-empty postfix expression
    """
    pass

evaluate_tests = [
    # case, expression, value
    ["3 * 4", [3, 4, "*"], 12],
    ["3 - 4", [3, 4, "-"], -1],
    ["3 - 4 * 5", [3, 4, 5, "*", "-"], -17],
    ["(3 - 4) * 5", [3, 4, "-", 5, "*"], -5],
    ["(3 - 4) * (5 - 6)", [3, 4, "-", 5, 6, "-", "*"], 1],
    ["no operation", [4], 4]
]

test(evaluate, evaluate_tests)
```

Answer

7.3 Queues

A stack is a sequence where items are added and removed from the same end. A queue is a sequence where items are added to one end and removed from the other.

7.3.1 Queues

People queue when boarding planes; cars queue at drive-ins. People and cars join the queue at the end and leave it at the front. Like stacks, queues are sequences of items ordered by arrival order, but contrary to stacks, the item arriving first is also the first to be processed: the ‘oldest’ item is at the bottom of the stack but at the front of the queue. A **queue** is a **first in, first out (FIFO)** sequence. The operations are very similar to a stack’s push, pop, and peek.

Operation	Effect	Algorithm in English
new	create an empty queue q	let q be an empty queue
length	the number of items in q	$ q $
enqueue	add an item to the back of q	add <i>item</i> to q
dequeue	remove the front item, if q isn’t empty	dequeue q
front	access the front item of, if q isn’t empty	front of q

Alternatively, the dequeue operation could remove and return the front item, but this wouldn't make the front operation redundant. Why?

With stacks, we can inspect the top item without 'disturbing' the stack, by popping it and pushing it immediately back. We can't inspect the front item of a queue with a dequeue followed by an enqueue operation: the front item would end up at the back of the queue.

Exercise 7.3.1

How would you represent a to-do list of tasks: with a general sequence, a stack, or a queue?

Hint Answer

Exercise 7.3.2 (optional)

Define the queue operations in the same way as the *stack operations*, using sequence notation. The operations are similar, so you may wish to define only one or two of them.

7.3.2 Queues with Python lists

Python's versatile lists can not only simulate stacks but also queues.

```
[1]: queue = []
queue.append("Alice")    # Alice arrives first
queue.append("Bob")
print("Next person served:", queue.pop(0))
queue.append("Clara")
print("Next person served:", queue.pop(0))
print("Next person served:", queue.pop(0))
print("People still waiting?", len(queue) > 0)

Next person served: Alice
Next person served: Bob
Next person served: Clara
People still waiting? False
```

This approach puts the front of the queue at index 0 and therefore uses `append` as the enqueue operation and `pop(0)` as the dequeue operation. The downside is that the latter takes time linear in the length of the list, because each remaining item is shifted one position down.

We could instead have the front of the queue at the last index. That would make dequeuing with `pop(-1)` take constant time, but enqueueing with `insert(0, item)` would take linear time.

To sum up, using Python lists and their methods makes one queue operation take linear time, which is fine if the queue never gets too long.

7.3.3 Queues with linked lists

The queue ADT can also be implemented with a linked list. If the front item is in the head node, then the dequeue operation removes the head node and the enqueue operation adds a node at the back of the linked list. We can traverse the linked list in linear time to find the last node and make it point to the new enqueued node, but by keeping a pointer to the last node we can enqueue a new node in constant time.

Likewise, we can compute the length of a queue in linear time, by traversing the linked list and counting nodes, but by keeping an integer with the current size we can return the length in constant time.

In both cases, we use a space-time tradeoff: storing additional data, and keeping it up to date as the queue operations are executed, leads to a better run-time. Here's the code.

```
[2]: # this code is also in m269_queue.py

class Queue:
    """A last-in, first-out sequence of objects, implemented with a
    linked list."""

    class Node:
        """A node in a linked list."""

        def __init__(self, item: object) -> None:
            """Initialise the node with the given item."""
            self.item = item
            self.next = None

    def __init__(self) -> None:
        """Initialise the queue to be empty."""
        self.head = None
        self.last = None
        self.length = 0

    def size(self) -> int:
        """Return the number of items in the queue."""
        return self.length

    def front(self) -> object:
        """Return the item at the front of the queue.

        Preconditions: self.size() > 0
        """
        return self.head.item

    def enqueue(self, item: object) -> None:
        """Add item to the back of the queue."""
        new_node = self.Node(item)
        if self.last is None:
            self.head = new_node
        else:
            self.last.next = new_node
        self.last = new_node
        self.length += 1
```

(continues on next page)

(continued from previous page)

```
node = Queue.Node(item)
if self.length == 0:
    self.head = node
    self.last = node
else:
    self.last.next = node
    self.last = node
self.length = self.length + 1

def dequeue(self) -> object:
    """Remove and return the item at the front of the queue.

    Preconditions: self.size() > 0
    """
    item = self.front()
    self.head = self.head.next
    if self.head == None:
        self.last = None
    self.length = self.length - 1
    return item
```

Here's the same example, using the class:

```
[3]: queue = Queue()
queue.enqueue("Alice") # Alice arrives first
queue.enqueue("Bob")
print("Next person served:", queue.dequeue())
queue.enqueue("Clara")
print("Next person served:", queue.dequeue())
print("Next person served:", queue.dequeue())
print("People still waiting?", queue.size() > 0)

Next person served: Alice
Next person served: Bob
Next person served: Clara
People still waiting? False
```



Info: In Java, queue operations are defined by interface `Deque`, which implements double-ended queues, an extension of queues. The interface is usually implemented by class `ArrayDeque` or `LinkedList`. Both the interface and the classes are in package `java.util`.

Exercise 7.3.3

If the queue were stored in the opposite order, with the last item in the head and the first item in the last node of the linked list, could we still implement the queue's operations in constant time?

Hint Answer

7.3.4 Using queues

Consider n children in a circle, numbered clockwise from 1 to n . Alice is in the centre. She points at the first child and starts reciting:

Eeny, meeny, miny, moe,
Catch a tiger by the toe.
If he hollers, let him go,
Eeny, meeny, miny, moe.

For each syllable, Alice points at a child, going clockwise. For example, with $n = 3$, she would point successively at children 1 (ee) 2 (ny) 3 (mee) 1 (ny) 2 (mi) 3 (ny) 1 (moe) for the first line. The child pointed to on the last syllable, the second ‘moe’, leaves the circle. The reciting and counting starts again on the next child. After going $n - 1$ times through the rhyme, one child is left in the circle. We want to know which child is that.



Info: This is a version of the Josephus problem.

Exercise 7.3.4

For this problem, we can represent a circle of children as a queue. How?

Hint Answer

Exercise 7.3.5

Given the number of children n , we want to know the number of the last remaining child. Outline an algorithm to compute that number.

Hint Answer

Exercise 7.3.6

Implement the algorithm you outlined by completing the following function, using the `Queue` class.

```
[4]: %run -i ../m269_queue  
  
from algoesup import test
```

(continues on next page)

(continued from previous page)

```

def counting_rhyme(n: int) -> int:
    """Return which child from 1 to n remains last in the circle.

    Preconditions: n > 0
    """
    pass

counting_rhyme_tests = [
    # case,           n,   last child
    ['1 child',     1,        1],
    ['2 children',  2,        1],
    ['3 children',  3,        2]
]

test(counting_rhyme, counting_rhyme_tests)

```

Hint Answer

Exercise 7.3.7

What is the complexity of the algorithm as implemented in the solution to the previous exercise?

Hint Answer

7.4 Priority queues

Families with young children often board an airplane before other passengers. Hospitals treat patients with more severe conditions first. To-do list apps list tasks with higher priority first.

All these are examples of queues that don't process items in a FIFO order. A **priority queue** is a sequence in which each item has a priority and items are removed from the queue from the highest to the lowest priority. In a **max-priority queue**, the highest priority corresponds to the largest value, whereas in a **min-priority queue** it is given by the smallest value. For example, if priorities are represented by positive integers, priority 1 is the highest priority in a min-priority queue and the lowest priority in a max-priority queue.

In the rest of this section we only implement unbounded max-priority queues, as bounded and min-priority queues are implemented very similarly. Actually, if priorities are given by integers, we get a max-priority queue to behave like a min-priority queue by negating the priorities when adding the items to the queue. Suppose item A has priority 1 and item B has priority 3. By inserting them with priorities -1 and -3 , the max-priority queue will return A before B, as if it were a min-priority queue.

Usually the order among items with the same priority is arbitrary, but a fairer priority queue keeps them in FIFO order.

Traditionally, priority queue operations are named differently from queue operations, to make clear which kind of queue we're using.

Operation	Effect	Algorithm in English
new	create an empty priority queue	let pq be a new priority queue
length	the number of items in pq	$ pq $
insert	add item i with priority p to pq	add (p, i) to pq
find max	obtain an item with highest-priority value	$\max(pq)$
remove max	remove the item obtained by find max	remove $\max(pq)$

The find and remove operations are only defined for non-empty priority queues.

7.4.1 With dynamic arrays: version 1

One simple way of implementing an unbounded max-priority queue is to use a dynamic array of priority-item pairs that are kept in ascending priority order. The insert operation appends a new priority-item pair and re-sorts the array. The find and remove operations simply access and remove the last pair.

Exercise 7.4.1

With dynamic arrays, the new and length operations take constant time. What is the complexity of the other operations for the approach outlined?

Operation	Complexity
insert	
find max	
remove max	

Hint Answer

In Python, we can use a list of tuples to represent the priority queue. Here's a simple example, with priorities from 1 (lowest) to 3 (highest).

```
[1]: tasks = [] # a priority queue
tasks.append(("go on holiday", 2))
tasks.sort()
tasks.append(("finish this chapter", 3))
tasks.sort()
tasks.append(("answer email", 1))
tasks.sort()
for times in range(len(tasks)): # noqa: B007
    print(tasks[-1][1], tasks[-1][0]) # print priority and task
    tasks.pop(-1)
2 go on holiday
3 finish this chapter
1 answer email
```

This is not quite working. The task with priority 3 should be listed first. Where's the error?

The tuples are put in ascending order by comparing them item by item (lexicographic order): first the task, then its priority. So the tasks end up in alphabetical order in the list and the task starting with 'g' is considered the highest-priority task.

How can we fix the issue?

Since the ordering is by priority, that must be the first item of the pair.

Here's a similar example to show the correct ordering.

```
[2]: tasks = [] # a priority queue
tasks.append((2, "go on holiday"))
tasks.append((1, "answer email"))
tasks.append((3, "finish this chapter"))
tasks.append((1, "remove old files"))
tasks.sort()
while tasks != []:
    task = tasks.pop(-1)
    print(task[0], task[1])
```

3 finish this chapter
2 go on holiday
1 remove old files
1 answer email

This example shows both the advantages and disadvantages of using Python lists instead of defining our own class. The advantages are a familiar notation and the flexibility of using list operations: we can sort only when needed, we can access and remove a highest-priority item with one operation, we can remove any other item if they leave the queue, etc.

One disadvantage is that this approach exposes the data structure and doesn't restrict the operations, which can lead to mistakes. Another disadvantage is that, by sorting the tuples, the approach assumes that the items in the queue are comparable, thus restricting its applicability.



Note: Try to restrict data types as little as possible.

Sorting priority-item pairs also leads to the priority queue not being fair. Among items with the same priority, the first to be removed is the one with highest-item value, not the one waiting longest. In the example, answering emails was added before but is done after the other priority 1 task, because of the alphabetical ordering.

7.4.2 With dynamic arrays: version 2

The solution to all these issues is to define our own class, with a bespoke insertion algorithm that is fair and only compares priorities.



Note: Using bespoke algorithms instead of general-purpose ones often leads to a better solution.

Here's an incomplete solution with separate lists (dynamic arrays) for the priorities and the items.

```
[3]: # this code is also in m269_priority.py

class ArrayPriorityQueue:
    """A dynamic array implementation of a fair max-priority queue.

    Items with the same priority are retrieved in FIFO order.
    """

    def __init__(self) -> None:
        """Create a new empty priority queue."""
        self.priorities = [] # in ascending order
        self.items = []

    def length(self) -> int:
        """Return the number of items in the queue."""
        return len(self.items)

    def find_max(self) -> object:
        """Return the oldest item with the highest priority.

        Preconditions: self.length() > 0
        """
        return self.items[-1]

    def remove_max(self) -> None:
        """Remove the oldest item with the highest priority.

        Preconditions: self.length() > 0
        """
        self.items.pop(-1)
        self.priorities.pop(-1)

    def insert(self, item: object, priority: object) -> None:
        """Add item with the given priority to the queue.
```

(continues on next page)

(continued from previous page)

```
Preconditions:  
- priority is comparable to the priorities of all existing  
→items  
"""  
index = 0  
# compute the index where to insert the item  
self.items.insert(index, item)  
self.priorities.insert(index, priority)
```

Exercise 7.4.2

Write an algorithm in English that finds the index of where to insert the item. Use variables *priorities* and *priority* and set a value for *index*. You can assume that priorities can be compared with $<$, \leq , \neq , etc.

Hint Answer

Exercise 7.4.3

Implement the algorithm in method `insert`. Run the following test, where `None` represents a patient without identification to make sure that the items are not comparable.

```
[4]: %run -i ../m269_test

hospital = ArrayPriorityQueue()  
hospital.insert("Bob", 1)  
hospital.insert("Alice", 3)  
hospital.insert(None, 1)  
for patient in ("Alice", "Bob", None): # this is the expected order  
    check("find_max", hospital.find_max(), patient, hospital.  
→length())  
    hospital.remove_max()
```

Answer

Exercise 7.4.4

Fill out the following table for the described approach.

Operation	Complexity
insert	
find max	
remove max	

Hint Answer

Exercise 7.4.5

Is it more efficient to have two lists for the items and the priorities, a single list with priority–item pairs, or does it not matter much? By more efficient I mean doing less work, not necessarily having better complexity.

Hint Answer

Exercise 7.4.6

There's yet another way of implementing a priority queue with dynamic arrays, with the following complexities:

Operation	Complexity
insert	$\Theta(1)$
find max	$\Theta(pq)$
remove max	$\Theta(pq)$

How? Outline the algorithm for each operation.

Answer

7.4.3 With a linked list

Version 2 of the dynamic array implementation can be adapted to linked lists. The head node has the item with the highest-priority value, i.e. the item returned by find max. To insert a new item, we start from the head node, skip all nodes with higher or equal priority, and insert a new node in the found place. With a linked list, no items are copied when inserting a new one: the insertion itself takes constant time once the insertion point is found, and there's no need to resize a linked list.

7.4.4 Min-priority queues

To finish this section, here's an example of using a max-priority queue as a min-priority queue.

Calendar apps allow us to attach a reminder to an event with a period of our choice, e.g. 15 minutes before the event. Reminders can be implemented as a priority queue, where an item is an event and its priority is the time when to issue the reminder. For example, the priority of a 3 pm meeting with a 30-minute reminder is 2:30 pm. The reminders are issued from earliest to latest time, so it's a min-priority queue, where the priority is the time.

Times can be represented as integers (minutes after midnight), so we can negate them to use our max-priority queue implementation. Here's an example, using an already fully implemented class.

```
[5]: %run -i ../m269_priority  
  
reminders = ArrayPriorityQueue()  
# meeting with 30 minute advance reminder
```

(continues on next page)

(continued from previous page)

```

reminders.insert("research group meeting @ 3pm", -(14 * 60 + 30))
# meetings with same reminder time
reminders.insert("M269 team meeting @ 11am", -(9 * 60))
reminders.insert("student supervision @ 10am", -(9 * 60))
while reminders.length() > 0:
    print(reminders.find_max())
    reminders.remove_max()

M269 team meeting @ 11am
student supervision @ 10am
research group meeting @ 3pm
    
```

Exercise 7.4.7

Why not model reminders with a FIFO queue?

Hint Answer

Exercise 7.4.8

The workaround for a max-priority queue to behave like a min-priority queue only works for integer priorities. Briefly explain how you would change our class's code to implement a min-priority queue for any comparable priority values.

Hint Answer

7.5 Summary

For some problems, it's useful to have sequences that restrict which items can be accessed and removed from the sequence.

In all of the following, operations to access and remove items are only defined for non-empty sequences.

7.5.1 Stacks

A **stack** is a **last-in, first-out (LIFO)** sequence: only the most recently added item can be accessed or removed. The stack ADT supports these operations, implemented by classes `Stack` and `LinkedListStack` in file `m269_stack.py` in your `notebooks` folder:

Operation	English	Python
new stack	let s be a new stack	<code>s = Stack()</code> or <code>s = LinkedListStack()</code>
size	$ s $	<code>s.size()</code>
push (add item)	push $item$ on s	<code>s.push(item)</code>
peek (last item added)	top of s	<code>s.peek()</code>
pop (remove last item)	pop s	<code>s.pop()</code>

When translating an algorithm from English to Python, we can combine the peek and pop operations when useful and write `item = s.pop()`.

The top item of a stack is stored in the last element of an array or in the first node of a linked list, in order to implement all operations in constant time.

Stacks can be used to evaluate postfix expressions, where operators are written after its operands. Contrary to infix expressions, where operators are written between operands, postfix expressions don't need parentheses to indicate the order of operations.

7.5.2 Queues

A **queue** is a **first-in, first-out (FIFO)** sequence: items are removed in the order they arrived.

The queue ADT provides the following operations, implemented by class `Queue` in file `m269_queue.py` in your notebooks folder:

Operation	English	Python
new	let q be an empty queue	<code>q = Queue()</code>
length	$ q $	<code>q.size()</code>
enqueue	add $item$ to q	<code>q.enqueue(item)</code>
dequeue	dequeue q	<code>q.dequeue()</code>
front	front of q	<code>q.front()</code>

All operations on queues take constant time, when implemented with a linked list that stores a pointer to the last item and stores the current size.

7.5.3 Priority queues

A **priority queue** is a sequence of items, each with an associated priority. Items are processed from highest to lowest priority value in a max-priority queue, and from lowest to highest priority value in a min-priority queue. Unless told otherwise, we can't assume in which order items with the same priority are processed.

Priorities can be of any data type with comparable values. For integer priorities, a max-priority queue can be used as a min-priority queue by negating the priorities when adding items.

A max-priority queue ADT provides the following operations, implemented by class `ArrayPriorityQueue` in file `m269_priority.py`.

Operation	English	Python
new	let pq be a new priority queue	<code>pq = ArrayPriorityQueue()</code>
length	$ pq $	<code>pq.length()</code>
insert	add $(priority, item)$ to pq	<code>pq.insert(item, priority)</code>
find max	$\max(pq)$	<code>pq.find_max()</code>
remove max	remove $\max(pq)$	<code>pq.remove_max()</code>

All operations take constant time except that inserting an item is linear in the length of the priority queue.

CHAPTER 8

UNORDERED COLLECTIONS

Sequences impose an order on the items: there's a first, a second, ..., a last item, even if we don't care about that order. This chapter introduces ADTs for collections of items that we don't need to keep ordered and shows how to implement them. *Chapter 10* will provide guidance on which ordered and unordered ADTs to use for which purpose.

This chapter supports the following learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter introduces two ADTs (maps and sets) and two array-based data structures (lookup and hash tables).

This chapter includes further examples to support these learning outcomes:

- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders.
- Analyse the complexity of algorithms to support software design choices.
- Write readable, tested, documented and efficient Python code.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

8.1 Maps

For many applications, we wish to retrieve data when we know only part of it. For example, we may wish to get an employee's full record by just their last name or office extension. This section introduces an ADT that allows us to handle data in such a way.

8.1.1 The map ADT

A **map** is an unordered collection of key–value pairs without duplicate keys. Each key–value pair is an item of the map. Different keys may be associated to the same value. The map ADT supports the following operations.

Operation	Effect	Algorithm in English
new	create a new empty map	let m be an empty map
size	the number of items in m	$ m $
membership	check if m has a given key	key in m
associate	associate a value to a key in m	let $m(\text{key})$ be value
lookup	obtain the value associated to a key	$m(\text{key})$
delete	remove a key–value pair from m	remove $m(\text{key})$
(in)equality	are two maps the same or different?	$m_1 = m_2$ or $m_1 \neq m_2$

The associate operation replaces the current value if the key is in the map, otherwise it creates a new key–value pair. The precondition of the lookup and delete operations is for the key to be in the map.

We assume that maps are **iterable**: it's possible to go through all keys with ‘for each *key* in *map*’. This allows us to discover the keys in the map to then access the values. As maps are unordered collections, the order in which keys are traversed isn't known in advance and may even change after adding or removing key–value pairs, depending on how the map is implemented.



Info: Maps are also called associative arrays. Some texts have separate operations to insert a pair and to replace a value instead of a single associate operation. A lookup is also called a search.

The above table uses functional notation for some operations because a map is, mathematically, a *function*: there's a set of inputs (the keys), a set of outputs (the values) and for each input there's a single output. Functions usually have a rule that computes the output for each input, whereas maps explicitly list all the inputs and their corresponding outputs. Most functions have infinite inputs, e.g. any positive integer, and hence correspond to infinite maps.

8.1.2 Using maps

Maps are widely used because they allow us to access an item by a mnemonic key, instead of an arbitrary position that is meaningless for unordered collections. Returning to the employee example, we could define several maps, all with essentially the same values (the employee records) but with different keys: last names, social security numbers, office phone numbers, etc. Depending on which piece of information the user has, they would use a different map to look up the employee. If there are several employees for the same key, then the associated value is a sequence of employees or some other type of collection.

Another example of a map is a bilingual dictionary: it associates each word in one language to one or more words in another language. Each key is a string and each value is a sequence of

strings. Here's a tiny Portuguese–English dictionary. Maps are best written as two-column tables.

Key	Value
‘alface’	(‘lettuce’)
‘carro’	(‘car’)
‘andar’	(‘floor’, ‘walk’)

Maps and other ADTs allow us to think about data in the most appropriate way for the problem at hand. The same data can be organised in different ways. The next two exercises provide an example.

Exercise 8.1.1

A max-priority queue is an ordered collection of priority–item pairs, ordered by priority. The priorities can be any comparable values. For this and the next exercise, assume that items with the same priority must be ordered by arrival, i.e. FIFO, order.

A priority queue can be seen as a map. What are the keys and values of the map?

Answer

Exercise 8.1.2

Using map operations, *outline* an algorithm for each operation: find max, remove max, add.

Hint Answer

Almost anything can be seen as a map. The next exercise is a reminder that just because we have a hammer, it doesn't mean that every problem is a nail.

Exercise 8.1.3

Booleans are hardly ever used as map keys. Why?

Answer

8.1.3 Lookup tables

The simplest way to implement a map is with a dynamic array of key–value pairs. With this approach, the new and size operations take constant time, but the membership, lookup, associate and delete operations are linear in the size of the map, as they have to do a linear search for the given key.

If a map's keys are natural numbers, we can use them to directly index the dynamic array. All map operations thereby take constant time. For example, a map from house numbers to sequences of strings, representing the residents of each house, can use this scheme. We initialise the dynamic array with, say, 10 empty sequences. When adding a new key–value pair, say Alice and Bob at house 50, we grow the dynamic array so that the largest index is 50, and then put string sequence (‘Alice’, ‘Bob’) at index 50.

If an array represents a map rather than a sequence, it's called a **lookup table**. Lookup tables are widely used to store pre-computed results. They're another example of a space–time tradeoff.

Although lookup tables can be dynamic arrays, they're usually static arrays, i.e. they're used when the map's keys form a fixed and relatively small collection, known in advance.

For example, consider a travel website that displays ever-changing forecast temperatures in degrees Celsius and Fahrenheit. They will all be within a certain range, e.g. -50 to 50 degrees Celsius. Using two lookup tables (mapping Celsius to Fahrenheit and vice versa) is more efficient than constantly re-applying the conversion formulas to the same temperature values.

In Python, one of the tables could be created like this:

```
[1]: FAHRENHEIT = [0] * 101
for celsius in range(-50, 51):
    FAHRENHEIT[celsius] = celsius * 9 // 5 + 32
FAHRENHEIT = tuple(FAHRENHEIT) # don't change table anymore
```

The first 51 positions have the Fahrenheit values for 0 to 50 degrees Celsius, and the last 50 positions have the values for -50 to -1 degrees Celsius. In programming languages that don't support negative indices, we would write `FAHRENHEIT[celsius + 50] = ...`.

What if the map's keys aren't integers? If we implement a **hash function** that converts each key to an index (this is called hashing the key), then we can still implement maps with lookup tables. Consider the following lookup table (in the form of a string) with a substitution cipher to encrypt text.

```
[2]: cipher = "lejqawntgckmfyrboxvhzsdiwu"
```

We must convert each lowercase English letter to an index from 0 to 25 to then look up the corresponding encrypted letter. Fortunately, this is easy with Python's `ord` function, which returns the Unicode code of a character.

```
[3]: def index_of(character: str) -> int:
    """Return a valid index for the character.

    Preconditions: character is a lowercase English letter
    Postconditions: the output is 0 for a, 1 for b, ..., 25 for z
    """
    return ord(character) - ord("a")
```

The hash function takes advantage of the lowercase letters being consecutive in the Unicode standard.

Any text can now be encrypted.

```
[4]: def encrypt(text: str, cipher: str) -> str:
    """Return the encrypted version of text, using cipher.

    Preconditions: len(cipher) = 26
    Postconditions: the output is text, with every lowercase letter
```

(continues on next page)

(continued from previous page)

```

replaced by the corresponding character in cipher
"""
encrypted = ""
for character in text:
    if "a" <= character <= "z":
        # apply hash function and lookup table
        encrypted = encrypted + cipher[index_of(character)]
    else:
        encrypted = encrypted + character
return encrypted

encrypt("This is top secret!", cipher)
[4]: 'Ttgv gv hrb vajxah!'

```

To sum up, if the keys aren't integers, then a map can be implemented with a lookup table together with a hash function that converts the keys to indices. The map operations will have the same complexity as the hash function, because once an index is obtained, accessing, replacing or deleting a value in the array takes constant time. (In a lookup table, deleting a value doesn't shift the others: the array represents a map, not a sequence.) Often, the hash function takes constant time, like `index_of`.



Info: Lookup tables are also called direct address (or addressing) tables.

Exercise 8.1.4

You've now seen two implementations of a priority queue: with a dynamic array of priority–item pairs sorted by priority ([Section 7.6](#)) and with a map of priorities to queues (exercises above).

Implementing an ordered collection (priority queue) with an unordered one (map) led to each operation searching for the highest key (priority) and thus taking time linear in the size of the map (number of priorities).

Operation	Sorted dynamic array	Map
find max	$\Theta(1)$	$\Theta(\text{priorities})$
remove max	$\Theta(1)$	$\Theta(\text{priorities})$
add	$\Theta(\text{priority queue})$	$\Theta(\text{priorities})$

These complexities assume that priorities aren't known in advance: they can be any comparable objects.

1. In which way may the complexities be different if we know the priorities in advance?
2. If we don't know the priorities in advance, is it always better to use sorted dynamic arrays?

Hint Answer

8.2 Dictionaries

Python's `dict` class implements a restricted form of maps. In M269, 'dictionary' (without any further qualification) refers to an object of type `dict`.



Info: TM112 introduces Python dictionaries in Block 3 Section 2.1. Some texts use 'dictionary' as a synonym for 'map'.

The operations are written in Python as follows, using the familiar list notation, but using keys instead of 0, 1, 2, etc. as 'indices':

Operation	Python
new	<code>d = dict()</code>
size	<code>len(d)</code>
membership	<code>key in d</code>
associate	<code>d[key] = value</code>
lookup	<code>d[key]</code>
delete	<code>d.pop(key)</code>

Like for lists, the `pop` method returns the associated value, and the negation of membership can be written `key not in d`.

We can represent the bilingual dictionary with a Python dictionary in which the keys are strings and the values are lists of strings.

```
[1]: pt_to_en = dict() # Portuguese to English dictionary
pt_to_en["alface"] = ["lattice"]
pt_to_en["alface"] = ["lettuce"] # replace wrong entry
pt_to_en["carro"] = ["car"]
pt_to_en["andar"] = ["floor", "walk"]
"carro" in pt_to_en
[1]: True
```

Dictionaries are iterable.

```
[2]: for word in pt_to_en: # iterate over the keys
    for translation in pt_to_en[word]:
        print(word, "means", translation)
alface means lettuce
carro means car
```

(continues on next page)

(continued from previous page)

```
andar means floor  
andar means walk
```

Python's implementation of maps guarantees that keys are iterated in the same order they were added or last updated, but you shouldn't rely on that in your M269 algorithms to keep them working with any implementation of the map ADT.

The `items` method returns a list-like object of tuples, one for each key–value pair. It's mostly used in for-loops.

```
[3]: for pair in pt_to_en.items():  
    word = pair[0]  
    for translation in pair[1]:  
        print(word, "means", translation)
```

```
alface means lettuce  
carro means car  
andar means floor  
andar means walk
```

Python allows the following shorthand notation.

```
[4]: for (word, translations) in pt_to_en.items():  
    for translation in translations:  
        print(word, "means", translation)
```

```
alface means lettuce  
carro means car  
andar means floor  
andar means walk
```

Dictionary literals are written as comma-separated pairs within curly braces. A colon separates each key from the corresponding value. Here's a shorter way of defining the bilingual dictionary.

```
[5]: pt_to_en = {  
    'alface': ['lettuce'],  
    'carro': ['car'],  
    'andar': ['floor', 'walk']  
}
```



Note: The empty dictionary can be written as `{ }`, but in M269 we use `dict()` instead, to avoid confusion with another data type, to be introduced later.

Dictionary keys may be integers that, unlike list indices, don't have to be consecutive. Here's a dictionary of addresses. The keys are the house numbers; the values are the residents' names.

```
[6]: our_houses = {23: "Alice", 45: "Bob"}
```

We can check if two dictionaries have the same key–value pairs or not with the (in)equality operators. In a dictionary, the key–value pairs are in no particular order.

```
[7]: our_street = {45: "Bob", 23: "Alice"}  
our_street == our_houses
```

```
[7]: True
```

```
[8]: our_street != {45: "Bob", 23: "Alissa"}
```

```
[8]: True
```

Like sequences, maps may be nested, i.e. the value associated to a key may be a map. For the bilingual dictionary, this could be used to distinguish the meanings of a word. For example, if ‘andar’ is used as a noun, then its translation is ‘floor’, whereas if ‘andar’ is used as a verb, then its translation is ‘walk’.

```
[9]: pt2en = {  
    'alface': {'noun': 'lettuce'},  
    'carro': {'noun': 'car'},  
    'andar': {'noun': 'floor', 'verb': 'walk'}  
}
```

We access inner dictionary values in the same way as nested list items.

```
[10]: inner_dictionary = pt2en["andar"]  
print(inner_dictionary["verb"])  
print(pt2en["andar"]["verb"]) # shorter alternative  
  
walk  
walk
```

8.2.1 Mistakes

Accessing or deleting a non-existent key raises an error.

```
[11]: pt_to_en["car"] # 'car' is among the values, not the keys  
-----  
→-----  
KeyError Traceback (most recent call last)  
Cell In[11], line 1  
----> 1 pt_to_en["car"] # 'car' is among the values, not the keys  
  
KeyError: 'car'
```

Dictionaries only retrieve data and check membership by key, not by value.

```
[12]: "carro" in pt_to_en
```

```
[12]: True
```

```
[13]: "car" in pt_to_en
```

```
[13]: False
```

Keys can't be added or removed while iterating over a dictionary.

```
[14]: roman_to_arabic = {"I": 1, "V": 5, "X": 10, "L": 50}
```

```
for (key, value) in roman_to_arabic.items():
    roman_to_arabic[key + "I"] = value + 1
```

```
-----  
RuntimeError
```

```
Traceback (most recent)
```

```
→call last)
```

```
Cell In[14], line 2
```

```
1 roman_to_arabic = {"I": 1, "V": 5, "X": 10, "L": 50}
```

```
----> 2 for (key, value) in roman_to_arabic.items():
    3     roman_to_arabic[key + "I"] = value + 1
```

```
RuntimeError: dictionary changed size during iteration
```

You may however change the values.

```
[15]: stock = {"trousers": 5, "t-shirt": 20, "dress": 12}
```

```
for (key, value) in stock.items(): # noqa: B007
    stock[key] = 0 # all sold out
```

```
stock
```

```
[15]: {'trousers': 0, 't-shirt': 0, 'dress': 0}
```

Dictionaries implement a restricted map ADT: keys can only be of types for which there's a hash function and that doesn't include lists and dictionaries. For example, consider a map of office building pairs to the names of their occupants. The keys can't be lists:

```
[16]: employee_by_location = { # occupants of each building's offices
```

```
    ["Main building", 4]: ["Alice", "Chan"],
    ["Annex", 3]: ["Bob"],
```

```
}
```

```
-----  
TypeError
```

```
Traceback (most recent)
```

```
→call last)
```

```
Cell In[16], line 1
```

```
----> 1 employee_by_location = { # occupants of each building's
    ↵offices
```

(continues on next page)

(continued from previous page)

```
2      ["Main building", 4]: ["Alice", "Chan"],  
3      ["Annex", 3]: ["Bob"],  
4 }
```

`TypeError: unhashable type: 'list'`

We get an error: lists are unhashable and thus the wrong type of key. The keys must be tuples:

```
[17]: employee_by_location = {  
        ('Main building', 4): ['Alice', 'Chan'],  
        ('Annex', 3): ['Bob']  
    }
```

Using a list or a dictionary as part of a key also leads to an error. For example, ('Bob', [1, 'Jan', 1970]) and ('Bob', {'day': 1, 'month': '1', 'year': 1970}) aren't valid Python keys, but ('Bob', '1 Jan 1970') and ('Bob', (1, 1, 1970)) are. Fortunately, most applications of dictionaries don't need complicated keys: integers, strings or tuples of both will suffice.

I explain in the next section why tuples are hashable but why lists aren't.

Exercise 8.2.1

Why can't we create a bilingual dictionary like this?

```
[18]: bilingual = dict()  
bilingual["alface"] = "lettuce"  
bilingual["carro"] = "car"  
bilingual["andar"] = "floor"  
bilingual["andar"] = "walk"
```

Answer

8.2.2 Using dictionaries

To further illustrate the dictionary operations, let's consider the problem of inverting a map, i.e. swapping keys and values, for bilingual dictionaries.

Function: invert

Inputs: *original*, a map with strings as keys and sequences of strings as values

Preconditions: true

Output: *inverted*, a map with strings as keys and sequences of strings as values

Postconditions: *inverted(word)* has *translation* if and only if *original(translation)* has *word*

The postconditions state that string *a* translates to string *b* in the inverted map if and only if *b* translates to *a* in the original map.

For testing I will use an empty map (edge case) and the Portuguese–English dictionary. The inversion of the former is the empty map; the inversion of the latter is:

Key (English)	Value (Portuguese)
‘lettuce’	(‘alface’)
‘car’	(‘carro’)
‘walk’	(‘andar’)
‘floor’	(‘andar’)

Unfortunately, this isn’t a very good test because the inverted dictionary doesn’t have multiple Portuguese translations for the same English word. Let’s add another translation of ‘floor’: ‘chão’. Since I have to change the Portuguese–English dictionary anyhow, I add an edge case for the value: an empty sequence, i.e. I add a Portuguese word but no English translation. Here are the two new dictionaries: the input and the expected output.

Key (Portuguese)	Value (English)	Key (English)	Value (Portuguese)
‘alface’	(‘lettuce’)	‘lettuce’	(‘alface’)
‘carro’	(‘car’)	‘car’	(‘carro’)
‘andar’	(‘floor’, ‘walk’)	‘walk’	(‘andar’)
‘chão’	(‘floor’)	‘floor’	(‘andar’, ‘chão’)
‘saudade’	()		

Exercise 8.2.2

Outline an algorithm that creates the right-hand side dictionary from the one on the left-hand side.

Hint Answer

Exercise 8.2.3

Implement the algorithm to replace `pass` in the function below. You don’t need to add further tests.

[19]: `from algoesup import test`

```
def invert(original: dict) -> dict:
    """Return the inverted dictionary.
```

In both dictionaries, the keys are strings and the values are lists of strings.

Postconditions:

word1 in output[word2] if and only if word2 in original[word1]
"""

(continues on next page)

(continued from previous page)

```

inverted = dict()
pass
return inverted

pt_to_en = {
    'carro': ['car'],
    'andar': ['floor', 'walk'],      # as in 'second floor'
    'chão': ['floor'],              # as in 'wooden floor'
    'saudade': []                  # translation omitted
}

en_to_pt = {
    'car' : ['carro'],
    'walk': ['andar'],
    'floor': ['andar', 'chão']
}

invert_tests = [
    #case,                      a_to_b,          inverted
    ('no words',                dict(),         dict()),
    ('pt_to_en',                 pt_to_en,       en_to_pt)
]

test(invert, invert_tests)

```

Answer

8.3 Hash tables

Maps can be implemented with *lookup tables* if the keys are known in advance, so that we can write a bespoke hash function that computes the index for each key. Different keys must be translated to different indices because each position of the lookup table only has one value of the map.

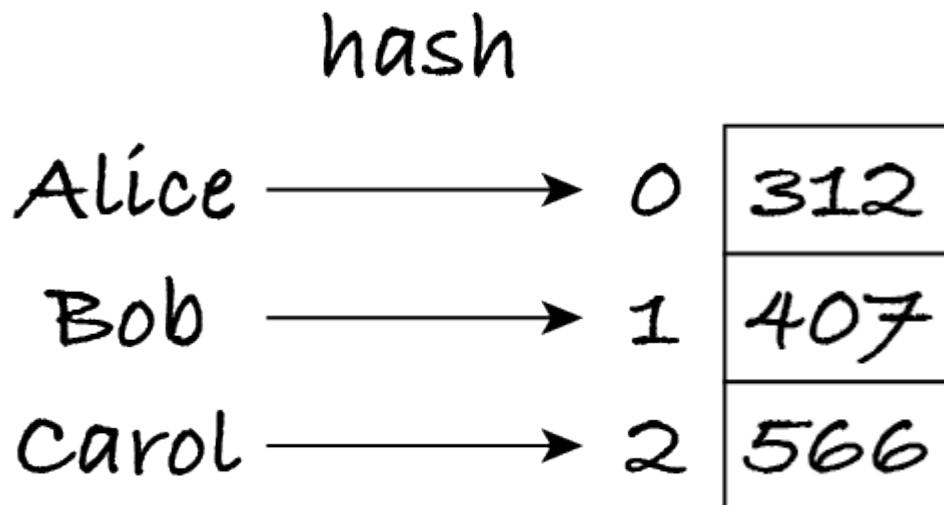
In a bilingual dictionary, any string can be a key (word of the language). When there's a potential infinite number of keys, it's impossible to guarantee that each associated value ends up in a different position of a finite table. A simple modification of the lookup table can handle this: we allow multiple key–value pairs per index.

8.3.1 With separate chaining

Consider a map of strings (names of employees) to integers (their phone extensions). If we know the employees' names, we can create a lookup table with as many entries as employees and a hash function that returns a distinct index for each employee. Adding, removing and replacing extensions involves obtaining the index and then putting, removing or replacing the integer at

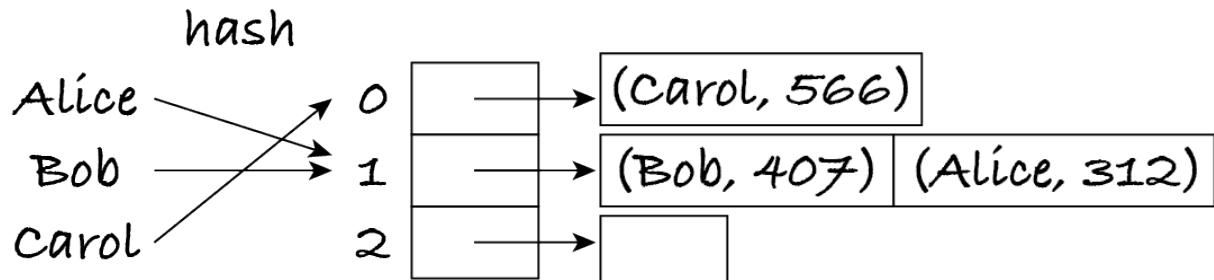
that position. Here's a small diagram for three employees Alice, Bob and Carol, for which the hash function returns 0, 1 and 2, respectively.

Figure 8.3.1



If instead the hash function has to return an index for *any* string, it may happen that for these particular strings the function returns 1 for 'Alice' and 'Bob' and 0 for 'Carol'. The lookup table then contains a sequence of employee-extension pairs at each position, with the empty sequence at index 2.

Figure 8.3.2



A **hash table with separate chaining** is a lookup table with a sequence of key–value pairs at each position of the table. Each sequence is called a **slot** of the hash table.



Info: Many authors use the term 'bucket' instead of 'slot'.

Here's the same hash table using Python lists.

```
[1]: [ # lookup table
      [('Carol', 599)], # first slot
       [('Bob', 407), ('Alice', 312)], # second slot
       [] # third slot
     ]
```

```
[1]: [[('Carol', 599)], [('Bob', 407), ('Alice', 312)], []]
```

Each map operation involves obtaining the index for the given key and then doing a linear search for that key in the slot at that index. If the key is in the map, it must be in *that* slot: no other slot has to be searched.

Exercise 8.3.1

The keys are used by the hash function to know which slot a value is in. Why do hash tables store the keys and not just the values?

Hint Answer

Ideally, we want each slot to be very short so that the linear search of a slot effectively takes constant time. If the table has length 1, there's a single slot with all key–value pairs. If the table has length 2, then each slot has half the pairs, assuming the hash function distributes them equally among the slots. If the table has length 3, each slot has (hopefully) one-third of the pairs, and so on. The longer the table, the shorter each slot is likely to be. The ratio between the number of pairs (size of the map) and the number of slots (size of the table) is the **load factor** of the hash table. The load factor is the average (mean) length of the slots. The example above has load factor $3 / 3 = 1$: three items for three slots. The mean length of each slot is 1. If there are 2 pairs and 4 slots, then the load factor and mean slot length are 0.5.

The lower the load factor, the higher the likelihood that each slot has at most one pair, which is ideal for performance. Unfortunately, the lower the load factor, the more empty slots the table has, which wastes memory. For example, a load factor of 0.1 means that 9 of 10 slots are empty, if the pairs are uniformly distributed. Space–time tradeoffs pop up everywhere in algorithms and data structures.

One way to get acceptable performance *and* memory usage is to use a dynamic array for the table and implement a growth-and-shrink policy that keeps the load factor within a desired range. For example, to keep the load factor between 0.5 and 1, we double the table length when adding a pair would make the load go over 1, and reduce the length when removing a pair would make the load drop below 0.5. Whenever the number of slots changes, the indices are recomputed for all keys to redistribute the pairs among the new (more or fewer) slots.



Info: Hash tables with open addressing use less memory than separate chaining because they have at most one key–value pair per position of the lookup table. However, the algorithms to implement the map operations are more complicated. M269 doesn't cover this variant of hash tables.



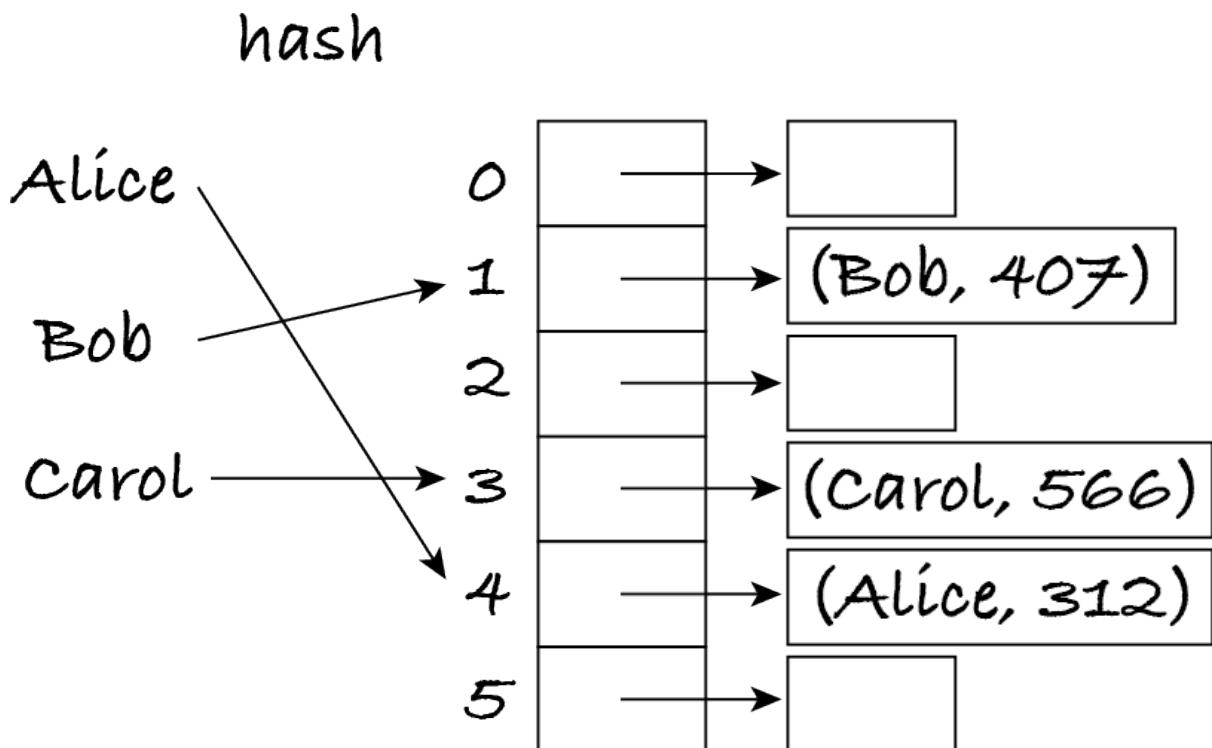
Info: Some texts use ‘dictionary’ as a synonym for ‘hash table’. In M269, a hash table is a data structure and dictionary is a Python data type.

8.3.2 Hash functions

If a hash function is to return an index for each key, it must know the length of the table. What happens in practice is that hash functions are generic, independent of the table size, and return a potentially very large integer h for the key. The map operations then transform h into a valid index. The easiest way is to use the modulo operation to obtain the remainder of h divided by the table length L . This results in an integer from 0 to $L - 1$.

Consider the earlier example and assume that h is 4 for ‘Alice’, 7 for ‘Bob’ and 3 and for ‘Carol’. With $L = 3$ we obtain the key-to-index assignment as shown earlier: ‘Alice’ and ‘Bob’ are put in slot $4 \bmod 3 = 7 \bmod 3 = 1$ and ‘Carol’ in slot $3 \bmod 3 = 0$. If the table is resized to $L = 6$, then ‘Alice’ and ‘Carol’ are expected to be in slots 4 and 3 respectively (their hash value h doesn’t change) and thus have to be put there.

Figure 8.3.3



When resizing a hash table we can't copy the slots wholesale to the same indices in the new static array, as we do for normal dynamic arrays: subsequent searches for keys would search the wrong slots and not find the keys! Therefore, when resizing a hash table, all indices have to be recomputed. This is not only useful to better redistribute the pairs when the table grows: it's essential for the map operations to work correctly.

Writing a hash function for a particular key type is dead easy. Here's one for strings:

```
def hash_string(string: str) -> int:
    return 0
```

It's a rubbish function of course. It will put all keys in the first position of the table and all map operations will take linear time in the size of the map.

Here's a slightly less worse hash function.

```
def hash_string(string: str) -> int:  
    return len(string)
```

This one puts all strings of the same length in the same slot. In most applications, the length of strings doesn't vary much. For example, if strings are English (or even German) words in common usage, then a hash table using this function would only have its first ~20 slots occupied.

This hash function is much better:

```
[2]: def hash_string(string: str) -> int:  
    """Return a hash number for the string."""  
    product = 1  
    for character in string:  
        product = product * ord(character)  
    return product
```

The function takes each character into account. Strings that differ by a single character have different hash values.

```
[3]: hash_string("algorithm")
```

```
[3]: 1885244446448213760
```

```
[4]: hash_string("algorythm")
```

```
[4]: 2172519790668893952
```

But strings that are permutations of each other have the same hash value.

```
[5]: hash_string("logarithm") # same as for 'algorithm'
```

```
[5]: 1885244446448213760
```

This will lead to a **collision**: the different keys will be put in the same slot.

The last function takes linear time in the length of the string. For short strings, this is equivalent to constant time, but if strings can have thousands of characters, e.g. they are whole documents, then a more efficient function may be needed.

Writing hash functions is easy; writing efficient hash functions that reduce collisions is hard. Hashing algorithms are quite mathematical and not covered in M269, but as always there's plenty of information online if you're curious.

Python has implemented hash functions for numbers, Booleans, strings and tuples. They can all be called via the generic `hash` function.

```
[6]: hash("algorithm")
```

```
[6]: -4858703937316695362
```

```
[7]: hash("logarithm")
```

```
[7]: -7962428355740776657
```

For security reasons, the hash values for strings are different between interpreter sessions. If you run the two cells above, you will get values different from mine; if you restart your notebook kernel and run the cells, you will get further different values.

8.3.3 Unhashable values

You may be wondering: if Python provides a hash function for tuples, why doesn't it provide one for lists?

```
[8]: hash((1, 2, 3))
```

```
[8]: 529344067295497451
```

```
[9]: hash([1, 2, 3])
```

```
-----  
TypeError                                     Traceback (most recent...  
    ↗call last)  
Cell In[9], line 1  
----> 1 hash([1, 2, 3])
```

```
TypeError: unhashable type: 'list'
```

Imagine I could use lists as keys, like this:

```
occupant = dict()  
office = ['Main building', 4]      # building and office number  
occupant[office] = 'Alice'
```

Lists are mutable, so I can change the name of the building: `office[0] = 'Headquarters'`. If I now try to obtain the occupant with the lookup operation (`occupant[office]`), what will happen?

The interpreter will compute the hash value of the office but since the office name changed, the hash value will almost certainly be different. The lookup operation will search for the office in the wrong slot, not find the key and raise an error.

If I try to look up the old key, the correct slot will be searched but since the key has changed, the linear search won't find a match and raise an error again.

In summary, if a key is modified after it was inserted in the hash table, then neither the old nor the new key can be found again, because the new key is in the slot computed for the old key. To avoid this problem, Python doesn't provide hash functions for lists and dictionaries, because they can be changed.

Some other languages do allow mutable values to be used as map keys. One way to achieve that is to put in the map a copy of the key, not a reference to the key object as Python does, so that any change to the key after the insertion does not affect the copy that is already in the map.



Info: Hash functions are also used in cryptography and to check if a file has been tampered with. Hash functions for encrypting or ‘fingerprinting’ files have different requirements than those for hash tables.

8.3.4 Complexity

Looking up, adding, removing or replacing a value in a map can take a while if any (or all!) of this happens:

- the hash function takes linear time in the size of a key and keys are large
- the hash function leads to many collisions
- the actual keys used in the application are somehow ‘skewed’ and lead to many collisions
- the resize policy is poor and leads to frequent reconstruction of the table.

Even if a hashing algorithm is linear in the size of the key, for the purposes of M269 we assume that keys have a small bounded length and thus we consider hashing to take constant time.

In the worst case, all keys end up in the same slot and every map operation takes linear time in the size of the map. In the best case, each slot has a small bounded number of key–value pairs and each map operation takes constant time.

We assume that the resize policy leads to constant time for each map operation.

To sum up, for most applications of maps, keys are short and general resize policies and hashing algorithms are good enough to obtain map operations with constant time. That’s why hash tables are a widely used ‘workhorse’ data structure, including for implementing Python’s dictionaries.



Info: Java’s class `java.util.HashMap` also implements maps using hash tables.

Like for sequences, the complexity of the equality operation is constant in the best case (when the maps have different sizes) and linear in the size of either map in the worst case, because the maps have to be compared one pair at a time.

8.3.5 Implementation

Here’s a partial implementation of a map, using a hash table with separate chaining.

```
[10]: class HashMap:  
    """An unordered collection of key-value pairs.  
    """
```

(continues on next page)

(continued from previous page)

```

Keys must be unique and hashable.
"""

# The hash table is a dynamic array of slots.
# Each slot is a dynamic array of key-value pairs.

def __init__(self) -> None:
    """Create an empty map."""
    self.slots = [[]] # start with 1 slot
    self.size = 0

def has(self, key: object) -> bool:
    """Return True if and only if key is in the map.

    Preconditions: key is hashable
    """
    index = hash(key) % len(self.slots)
    slot = self.slots[index]
    # linear search of the key in the only slot it can be
    for pair in slot:
        if pair[0] == key:
            return True
    return False

def grow(self) -> None:
    """Grow the dictionary if necessary.

    Postconditions:
    if pre-self has load factor 1, post-self has load factor 0.5
    """
    capacity = len(self.slots)
    if self.size == capacity:
        # new hash table with double the slots, all empty
        new_capacity = capacity * 2
        new_slots = []
        for each_slot in range(new_capacity): # noqa: B007
            new_slots.append([])
        # put each pair in the correct slot in the new table
        for slot in self.slots:
            for pair in slot:
                index = hash(pair[0]) % new_capacity
                new_slots[index].append(pair)
        # use the new hash table
        self.slots = new_slots
    
```

(continues on next page)

(continued from previous page)

```
def associate(self, key: object, value: object) -> None:
    """Associate value to key in the map.

    Preconditions: key is hashable
    Postconditions: looking up key in self returns value
    """

    self.grow()
    index = hash(key) % len(self.slots)
    slot = self.slots[index]
    for pair in slot:
        if pair[0] == key:
            pair[1] = value
            return
    slot.append([key, value])
    self.size = self.size + 1
```

Let's have a peek at how the hash table changes as new pairs are added. The following constructs a hash table of Unicode codes to characters.

```
[11]: characters = HashMap()
for letter in "algorithm":
    characters.associate(ord(letter), letter)
    print(characters.slots)
for letter in "moralgith": # test membership operation
    if not characters.has(ord(letter)):
        print("error: missing", letter)

[[[97, 'a']]]
[[[108, 'l']], [[97, 'a']]]
[[[108, 'l']], [[97, 'a']], [], [[103, 'g']]]
[[[108, 'l']], [[97, 'a']], [], [[103, 'g']], [111, 'o']]
[], [[97, 'a']], [[114, 'r']], [], [[108, 'l']], [], [], [[103, 'g'],
→], [111, 'o']]]
[], [[97, 'a'], [105, 'i']], [[114, 'r']], [], [[108, 'l']], [], [],
→ [[103, 'g'], [111, 'o']]]
[], [[97, 'a'], [105, 'i']], [[114, 'r']], [], [[108, 'l'], [116, 't'],
→]], [], [], [[103, 'g'], [111, 'o']]]
[[[104, 'h']], [[97, 'a'], [105, 'i']], [[114, 'r']], [], [[108, 'l'],
→], [116, 't']], [], [], [[103, 'g'], [111, 'o']]]
[], [[97, 'a']], [[114, 'r']], [], [[116, 't']], [], [], [[103, 'g',
→]], [[104, 'h']], [[105, 'i']], [], [[108, 'l']], [[109, 'm'],
→]], [], [[111, 'o']]]
```

Note how pairs move around as the table grows. For example, 'l' and its code move across slots 0, 4 and 12. Pairs that are in the same slot for a while, like 'g' and 'o', end up in separate slots when the table gets larger.

Exercise 8.3.2

Some Python interpreters also store the hash value of each key in the table, i.e. each slot has hash–key–value triples rather than key–value pairs. Can you think why?

Hint Answer

Exercise 8.3.3

Suppose I didn't implement the method to grow the hash table and instead initialised the table with one thousand slots. Would this map implementation still work if the load factor goes above 1?

Hint Answer

8.4 Sets

A **set** is an unordered collection of unique items, i.e. without duplicates. The items in a set are traditionally called its elements or members. In mathematics a set is written with curly braces. For example, $\{1, 2, \text{'hi'}\}$ and $\{\text{'hi'}, 2, 1\}$ are the same set, as the order in which we list set members doesn't matter.

8.4.1 The set ADT

The set ADT supports the following operations:

Operation	Effect	Maths	English
new	create new set	let s be $\{\}$	let s be the empty set
size	the number of elements	$ s $	$ s $
membership	check if item i is in s	$i \in s$	i in s
add	add an item i to s		add i to s
remove	take an item i from s		remove i from s
intersection	the items in $s1$ and in $s2$	$s1 \cap s2$	intersection of $s1$ and $s2$
union	the items in $s1$ or in $s2$	$s1 \cup s2$	union of $s1$ and $s2$
difference	the items in $s1$ but not in $s2$	$s1 - s2$	$s1 - s2$

Some examples of the above operations:

- $\{1, 2, 3\} \cup \{4, 5, 2\} = \{1, 2, 3, 4, 5\}$
- $\{1, 2, 3\} - \{4, 5, 2\} = \{1, 3\}$
- $\{1, 2, 3\} \cap \{4, 5, 2\} = \{2\}$
- $\{1, 2, 3\} \cap \{4, 5, 6\} = \{\}$

Two sets are said to be **disjoint** if they have no common elements: their intersection is the empty set.

Adding an item corresponds to doing ‘let s be $s \cup \{i\}$ ’. Removing an item corresponds to doing ‘let s be $s - \{i\}$ ’. Adding an already existing item or removing an nonexistent item has therefore no effect on the set.

The set ADT also includes comparison operations. Two sets can be compared for (in)equality, e.g. $\{1, 2, 3\} = \{3, 1, 2\}$ but $\{1, 2, 3\} \neq \{3, 1, 4\}$.

A set A is a **subset** of B , and B a **superset** of A , written $A \subseteq B$ or $B \supseteq A$, if every item of A is also in B . Set A is a **proper** subset of B , written $A \subset B$ (or B is a proper superset of A , written $B \supset A$), if $A \subseteq B$ and $A \neq B$. For example, $\{1, 2\} \subset \{1, 2, 3\}$.



Info: The size of a set is also known as its cardinality. The difference operation is also written $s1 \setminus s2$. The empty set is also written as \emptyset . MST124 Unit 3 Section 1.1 introduces sets of real numbers and set notation.

8.4.2 Sets in Python

Python has a built-in class `set` to represent sets. Set literals are written as comma-separated items within curly braces, e.g. `{1, 2, 3}`. Python sets are iterable. The operations are written as follows.

Operation	Python
new	<code>s = set()</code>
size	<code>len(s)</code>
membership	<code>item in s</code> or <code>item not in s</code>
add	<code>s.add(item)</code>
remove	<code>s.discard(item)</code>
union	<code>s1.union(s2)</code>
intersection	<code>s1 & s2</code> or <code>s1.intersection(s2)</code>
difference	<code>s1 - s2</code> or <code>s1.difference(s2)</code>

The union operation can also be written as `s1 | s2`.



Note: In Python, `{}` represents the empty dictionary, not the empty set, so always use `set()` instead.

The last three operations create a new set: they don’t modify either input set. Here’s a simple example with positive integers.

```
[1]: odd = {1, 3, 5}
      even = {2, 4, 6}
      prime = {2, 3, 5}
```

(continues on next page)

(continued from previous page)

```

print("all:", odd | even)
print("even primes:", even & prime)
print("odd primes (primes that aren't even):", prime - even)
print("even numbers that aren't prime:", even - prime)

all: {1, 2, 3, 4, 5, 6}
even primes: {2}
odd primes (primes that aren't even): {3, 5}
even numbers that aren't prime: {4, 6}
    
```

Note that the IPython interpreter displays set members in sorted order. Internally, the items may be stored in any order, e.g. `odd | even` may be stored as `{1, 3, 5, 2, 4, 6}`.



Note: Your algorithms on sets must not rely on any particular order of the items.

Exercise 8.4.1

Write alternative expressions for the even primes and the odd primes, without using the set `even`.

Hint Answer

It's possible to build expressions from these operations. Their associativity and precedence in relation to other operations is as follows, with highest precedence at the top of the table.

Operators	Associativity
arithmetic	left (except for exponentiation and negation)
intersection	left
union	left
comparison and membership	left
logical	left (except negation)

The set difference operator has the same precedence and associativity as arithmetic difference (subtraction). The set membership operator has the same precedence and associativity as membership for other collections. In the next example, ‘number’ refers to a positive integer.

```

[2]: print("all:", odd | even | prime)
print("odd numbers that aren't even primes", odd - (prime & even))
print("non-prime odd numbers that are even:", odd - prime & even)
print("prime numbers that are odd or even", (odd | even) & prime)
print("numbers that are even primes or odd", odd | even & prime)

all: {1, 2, 3, 4, 5, 6}
odd numbers that aren't even primes {1, 3, 5}
non-prime odd numbers that are even: set()
    
```

(continues on next page)

(continued from previous page)

```
prime numbers that are odd or even {2, 3, 5}  
numbers that are even primes or odd {1, 2, 3, 5}
```

Like for Boolean expressions, it's best to always put parentheses to show our intentions, e.g. `(odd - prime) & even` for the second expression.



Note: Write all parentheses in set expressions, even if they're redundant.

The set comparison operators are written like the arithmetic comparisons.

```
[3]: print("are all primes odd?", prime <= odd)  
print("are all odd numbers prime?", odd <= prime)  
print("are some numbers not prime?", prime < odd | even)  
  
are all primes odd? False  
are all odd numbers prime? False  
are some numbers not prime? True
```

The last expression asks the equivalent question: are the prime numbers a proper subset of all numbers?

Any sequence can be converted to a set of its unique items, using the type constructor.

```
[4]: set([3, 4, 3, 6, 2, 1, 6])  
[4]: {1, 2, 3, 4, 6}
```

This is a shorter way of writing:

```
[5]: unique = set()  
for item in [3, 4, 3, 6, 2, 1, 6]:  
    unique.add(item)  
unique  
[5]: {1, 2, 3, 4, 6}
```

8.4.3 Implementing sets

The set ADT can be implemented with a sequence data type, but that makes adding an item take linear time, to check if it's already there.

A set can be seen as a map from items to Booleans, stating for each item if it's a member of the set. Therefore, any map implementation can be used to implement sets. For example, if the potential set members are limited and known in advance, we can use a lookup table of Booleans. Set operations like intersection are easy to implement by going through two lookup tables and applying Boolean operations.

Python's sets are implemented with hash tables and thus items must be hashable. Like dictionaries, sets aren't hashable themselves and so can't be used as keys.



Info: In Java, the interface `Set` defines the set data type. Class `HashSet` implements the interface with a hash table. Both the interface and class are in package `java.util`.

The add, remove and membership operations take constant time for Python sets. As for the operations on two sets a and b , union has complexity $\Theta(|a| + |b|)$, intersection has complexity $\Theta(\min(|a|, |b|))$, i.e. is linear in the smallest of both sets, and the difference $a - b$ is linear in the size of the first set: $\Theta(|a|)$.

Exercise 8.4.2

Explain the complexities of the union, intersection and difference operations.

Hint Answer

Exercise 8.4.3

Checking if two sets are disjoint can be done with the Boolean expression `len(s1 & s2) == 0`.

1. Why isn't this an efficient way of checking disjointness?
2. Outline a better algorithm.
3. Explain why it's better, by comparing the memory use and the best- and worst-case complexities against those of the expression.



Note: The shortest algorithm is not necessarily the most efficient.

Hint Answer

8.4.4 Using sets

An efficient implementation of a set is very useful: contrary to lists, stacks and queues, it supports the add, remove and membership operations in constant time, for *every* item. Here's a problem that uses a set just as a basic collection of items.

Exercise 8.4.4

A computing society is organising a programming contest for schools. Each school can send up to 4 teams of students. Each team is identified by a string like 'DS2', with the team's number after the school's initials. The best team of each school gets a certificate. Given the final team ranking, compute the teams that get a certificate. Add tests.

```
[6]: from algoesup import test

def certificates(ranking: list) -> list:
    """Return the best team of each school.

    The input and output are lists of strings (team names).
    Each string is the name of a school and a digit from 1 to 4.

    Preconditions:
    - ranking is a non-empty list ordered from first to last team
    - there are no duplicate teams
    Postconditions:
    - the output has the first team in 'ranking' of each school
    - the output strings are in the same order as in 'ranking'
    """
    best_teams = []
    pass
    return best_teams

certificates_tests = [
    # case,           ranking,                  certificates
    ('3 schools',   ['C1', 'B2', 'B1', 'A1', 'C2'], ['C1', 'B2', 'A1']),
    # new tests:
]
test(certificates, certificates_tests)
```

Hint Answer

8.5 Summary

8.5.1 Maps and dictionaries

A **map** is an unordered collection of **key–value pairs** with unique keys. The same value may be associated with different keys. Python's `dict` data type implements the map ADT.

Operation	English	Python	Complexity (best/worst)
new	let m be an empty map	<code>d = dict()</code>	$\Theta(1)$
size	$ m $	<code>len(d)</code>	$\Theta(1)$
membership	$key \in m$	<code>key in d</code>	$\Theta(1)$
associate	let $m(key)$ be $value$	<code>d[key] = value</code>	$\Theta(1)$
lookup	$m(key)$	<code>d[key]</code>	$\Theta(1)$
remove	remove $m(key)$	<code>d.pop(key)</code>	$\Theta(1)$
equality	$m_1 = m_2$	<code>d1 == d2</code>	$\Theta(1) / \Theta(m_1)$
inequality	$m_1 \neq m_2$	<code>d1 != d2</code>	$\Theta(1) / \Theta(m_1)$

Accessing a value raises an error if the dictionary doesn't contain the given key.

Dictionaries are implemented with hash tables, described below. Dictionaries take more memory than sequences with the same key–value pairs. We assume all dictionary operations take constant time, except (in)equality, which takes linear time in the size of the dictionary in the worst case.

Dictionaries are iterable: `for key in a_dict` iterates over all keys in `a_dict` and `for (key, value) in a_dict.items()` iterates over all key–value pairs. While iterating over a dictionary, no key–value pair can be added or removed. Iterating over a dictionary's keys or items takes linear time in the size of the dictionary.

8.5.2 Lookup and hash tables

A map with natural numbers as keys can be implemented with a **lookup table**, an array in which the indices are the keys and the items are the values. If the keys are characters, then Python's function `ord` can be used to return their Unicode code, which is a natural number. Lookup tables are often used to store pre-computed values.

A **hash table with separate chaining** is a lookup table of sequences of key–value pairs. Each sequence is called a **slot**.

The **load factor** is the number of pairs (size of the map) divided by the number of slots (size of the table), i.e. the mean number of pairs per slot. The lower the load factor, the more memory is used, but the higher the chance that each slot has at most one pair. If two different keys are associated to the same slot, a **collision** occurs. With separate chaining, the collision resolution algorithm simply adds both keys to the same slot.

The hash table is implemented with a dynamic array, to increase the number of slots as the dictionary size increases and keep a low load factor. When the table grows or shrinks the slots of all pairs have to be recomputed.

To search, add, replace or delete a value by key, we compute for the given key the slot it must be in, and then do a linear search of the key in that slot. With a low load factor, a hash function that reduces collisions, and short keys, map operations take constant time, which we assume is the usual situation. In the worst case (all key-value pairs in the same slot), operations are linear in the size of the map.

In Python, lists and dictionaries aren't **hashable**, i.e. can't be used as keys, to avoid inadvertently changing a key after it was inserted in the dictionary. A tuple is hashable only if all its items are.

Python has a built-in hash function, named `hash`.

8.5.3 Sets

Sets are unordered collections without duplicate items. Python's `set` class is implemented like a dictionary and thus requires items to be hashable. Python's sets are iterable but not hashable.

Operation	Maths	English	Python	Complexity (best/worst)
new	let s be $\{ \}$	let s be the empty set	<code>s = set()</code>	$\Theta(1)$
size	$ s $		<code>len(s)</code>	$\Theta(1)$
membership	$i \in s$	i in s	<code>item in s or item not in s</code>	$\Theta(1)$
add		add i to s	<code>s.add(item)</code>	$\Theta(1)$
remove		remove i from s	<code>s.discard(item)</code>	$\Theta(1)$
union	$s_1 \cup s_2$	union of s_1 and s_2	<code>s1.union(s2)</code>	$\Theta(s_1 + s_2)$
intersection	$s_1 \cap s_2$	intersection of s_1 and s_2	<code>s1 & s2 or s1.intersection(s2)</code>	$\Theta(\min(s_1 , s_2))$
difference	$s_1 - s_2$		<code>s1 - s2 or s1.difference(s2)</code>	$\Theta(s_1)$
(proper) subset	$s_1 \subset s_2$ and $s_1 \subseteq s_2$		<code>s1 < s2 and s1 <= s2</code>	$\Theta(1) / \Theta(s_1)$
(proper) superset	$s_1 \supset s_2$ and $s_1 \supseteq s_2$		<code>s1 > s2 and s1 >= s2</code>	$\Theta(1) / \Theta(s_2)$
(in)equality	$s_1 = s_2$ and $s_1 \neq s_2$		<code>s1 == s2 and s1 != s2</code>	$\Theta(1) / \Theta(s_1)$

Set union can also be written as `s1 | s2`.

Two sets are **disjoint** if their intersection is empty.

CHAPTER 9

PRACTICE 1

This chapter introduces no new concepts: it only provides problems to practise for the TMA. In previous exercises, you knew in advance which data type to use; the main aim of this chapter is to practise choosing appropriate types, and using a combination of types when necessary. You should skim the advice in the *next chapter* before attempting the problems.

The problems are in increasing difficulty order, but that's only my opinion: your mileage may vary. I recommend you first read through all the problems to decide which to attempt and in what order. I suggest you do one each of the easier (Pangram and Election), medium (Voucher and Trains) and harder (Browsing and SMS) problems. If you still have time left, attempt the remaining problems. Attempt all non-optional exercises before attempting any of the optional ones. Don't worry if you don't finish all the problems: your priority is to complete and submit your first TMA next week.

Every problem in this chapter can be solved in various ways: the solutions I present aren't exhaustive. If you find a different approach, I encourage you to post it in the forums. Please provide spoiler alerts for your fellow students who haven't solved the problem yet.

Note that my answers are often more verbose than expected in a TMA as they also include alternatives, notes and the thinking process.

You can freely discuss the problems with your tutor, in tutorials and on the forums. If you have a study buddy, you may wish to think independently about a problem, discuss it together, implement different or similar solutions separately, and finally review each other's code.

Working together has several advantages: you may come up with solutions you wouldn't have thought of if working alone; you practise explaining algorithms to other people; you get an extra pair of eyes to look at your code and suggest improvements.

This chapter's problems support the following learning outcomes.

- Develop and apply algorithms and data structures to solve computational problems – you will apply several of the ADTs and techniques you learned.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – you will be asked to outline your approach, rather than implement it.

- Analyse the complexity of algorithms to support software design choices – you will have to suggest or evaluate alternative approaches.
- Write readable, tested, documented and efficient Python code – you will have to add tests and use Python data types and operations.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

9.1 Pangram

A pangram is some text containing all letters of the alphabet. Pangrams are used to illustrate fonts, to check that keyboards work, to test cryptographic protocols, etc. The most famous pangram in English is ‘The quick brown fox jumps over the lazy dog’. Given a string, find which letters must be added for it to become a pangram.

Here’s the code for you to complete. Read it and then go through the questions below.

```
[1]: from algoesup import test

def missing_letters(text: str) -> str:
    """Return all English alphabet letters that aren't in text.

    Preconditions: all letters in text are lowercase
    Postconditions: the output are the letters from a to z that
    don't occur in text, in alphabetical order
    """
    pass

PANGRAM = "the quick brown fox jumps over the lazy dog."

missing_letters_tests = [
    # case,           text,                  missing
    ['pangram',      PANGRAM,              ''],
    ['no vowels',    'bcd fgh jklmn pqrst vwxyz', 'aeiou'],
    # new tests:
]

test(missing_letters, missing_letters_tests)
```

In the following exercises, assume the input and output variables are called *text* and *missing*, as in the test table comment above.

9.1.1 Problem definition

Exercise 9.1.1

Which further problem instances should be tested?

Case	text	missing

Hint Answer

9.1.2 Algorithm and complexity

Exercise 9.1.2

Outline (or write, step by step) two different algorithms to solve the problem and the ADT(s) they use.

Hint Answer

Exercise 9.1.3

What are the best- and worst-case complexities of each algorithm?

Hint Answer

Exercise 9.1.4

Which algorithm do you think is the more efficient one? Take into account the complexity, the actual operations carried out and the memory used by each algorithm.

Hint Answer

9.1.3 Code and tests

Exercise 9.1.5

Implement and test your chosen approach in the code cell above. Add tests according to your answer to the first exercise.

Answer

Exercise 9.1.6 (optional)

Implement the other algorithm and compare the run-times.

Hint

9.2 Election

Given a list of votes, each vote being the name of a candidate, which candidate won the election? The votes are in the order they were cast. If two or more candidates are tied, a second round is needed.

```
[1]: from algoesup import test

def winner(votes: list) -> str:
    """Return the most frequent string in votes, a list of strings.

    Preconditions:
    - votes isn't empty and doesn't have the string 'round 2'
    Postconditions:
    - return 'round 2' if two or more strings are equally frequent
    """
    pass

winner_tests = [
    # case,           votes,           name
    ['2 of 2 tied', ['Alice', 'Bob', 'Bob', 'Alice'], 'round 2'],
    ['1 of 2 wins', ['Alice', 'Bob', 'Alice', 'Alice'], 'Alice'],
    ['1 of 3 wins', ['Bob', 'Chan', 'Chan', 'Alice'], 'Chan'],
    # new tests:
]

test(winner, winner_tests)
```

9.2.1 Problem definition

Exercise 9.2.1

Which further problem instances should be tested?

Case	votes	name

Hint Answer

9.2.2 Algorithm and complexity

Exercise 9.2.2

Outline an algorithm and its ADT(s) to solve this problem.

Hint Answer

Exercise 9.2.3

What is the worst-case complexity of your algorithm?

Hint Answer

Exercise 9.2.4

Consider the following outline of an algorithm:

Go through the votes once, to count them for each candidate. Sort the candidate–vote pairs in decreasing order of their votes. If there's only one candidate, that's the winner. Otherwise, check if the first two candidates are tied.

Is it more efficient than yours?

Hint Answer

9.2.3 Code and tests

Exercise 9.2.5

Implement and test your approach. Add tests according to your answer to the first exercise.

Hint Answer

9.3 Voucher

Given a list of products in a store and their prices, we want to know all product pairs that can be bought by spending exactly a certain amount.

```
[1]: from algoesup import test

def pairs(store: list, voucher: int) -> set:
    """Return all pairs of products that together cost voucher.

    A product is represented by a string-integer tuple:
    the product's name and its positive price.
    Store is a list of products.
    The output is a set of product pairs (tuples).

    Preconditions: voucher > 0
```

(continues on next page)

(continued from previous page)

```
Postconditions: for each (p1, p2) in the output,
- p1 and p2 occur in store
- p1 doesn't occur after p2 in store
- the price of p1 + the price of p2 == voucher
"""
results = set()
pass
return results

F1 = ("food", 1)
F2 = ("food", 2)
F5 = ("food", 5)
F6 = ("food", 6)
T5 = ("toy", 5)

pairs_tests = [
    # case,           store,          voucher, results
    ('0 products',   [],             5,         set()),
    ('1 product',    [F5],          5,         set()),
    ('no pair',      [F1,F5,F2,F6], 9,         set()),
    ('1 pair',       [F1,F5,F2,F6], 3,         {(F1,F2)}),
    ('2 pairs',      [F1,F5,F2,F6], 7,         {(F1,F6), (F5, F2)}),
    ('same price',   [F1,F5,F2,T5], 10,        {(F5,F5), (F5,T5), (T5,
    ↵T5)}) )
]

test(pairs, pairs_tests)
```

9.3.1 Algorithm and complexity

Exercise 9.3.1

Outline an algorithm and its ADT(s) to solve this problem in $\Theta(|store|)$ time in the best case.

Hint Answer

Exercise 9.3.2

Explain why your algorithm has the desired best-case complexity.

Hint Answer

Exercise 9.3.3

What's the worst-case complexity of your algorithm?

Hint Answer

9.3.2 Code and tests

Exercise 9.3.4

What's the best way of implementing the ADT used?

Hint Answer

Exercise 9.3.5

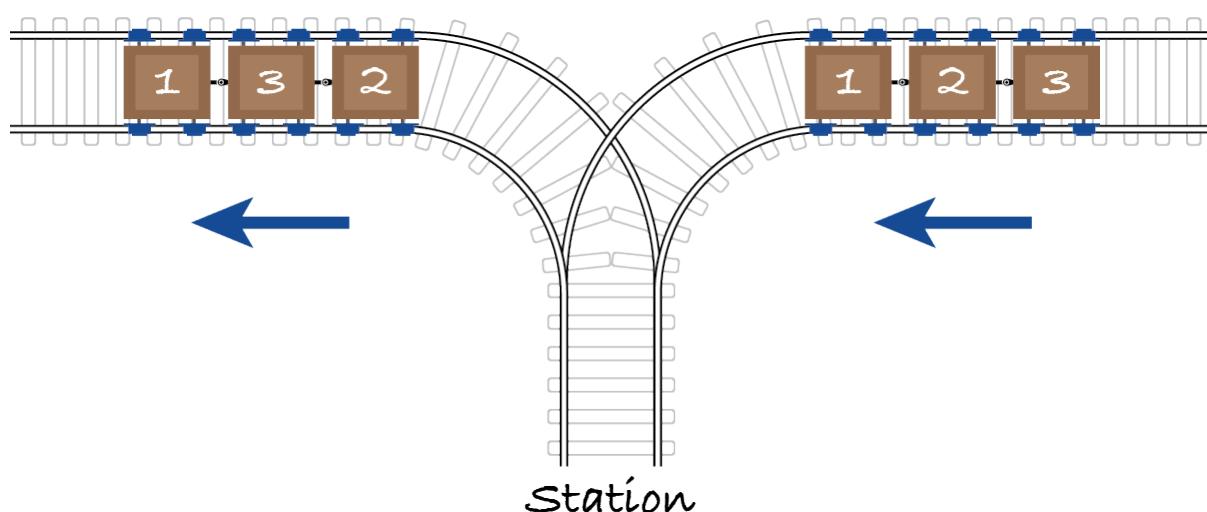
Implement and test your approach. You don't need to add tests.

Answer

9.4 Trains

Consider a track layout in the shape of the letter Y.

Figure 9.4.1



A train formed of wagons numbered 1, 2, 3, ..., n comes from the east into the station and needs to leave it to the west with the wagons in a possibly different order. Each wagon can be detached from the others and the station has capacity for all n wagons. Once a wagon is in the station, it can't be moved back to the east. Once a wagon is in the west track, it can't be moved back to the station.

The example in the figure is a train with 3 wagons that must leave in the order 1, 3, 2. The following operations achieve that order:

- move wagon 1 from the east into the station and out to the west
- move wagon 2 into the station

- move wagon 3 into the station and out
- move wagon 2 out of the station.

The station master wants to know, given the number of wagons and their outgoing order, if it's possible to achieve that order.

```
[1]: from algoesup import test

def rearrange(wagons: int, outgoing: list) -> bool:
    """Check if the incoming train can be rearranged into outgoing.

    Preconditions:
    - wagons > 0
    - outgoing is a permutation of the numbers from 1 to wagons
    """
    pass

rearrange_tests = [
    # case,           wagons,  outgoing,      rearrange?
    ['keep order',   3,       [1, 2, 3],     True],
    ['invert',       3,       [3, 2, 1],     True],
    ['swap',         3,       [1, 3, 2],     True],
    ['move to front', 5,       [5, 1, 2, 3, 4], False],
    # new tests:
]
test(rearrange, rearrange_tests)
```

9.4.1 Problem definition

Exercise 9.4.1

What kind of problem is this?

Answer

Exercise 9.4.2

Which further problem instances should be tested?

Case	wagons	outgoing	rearrange?

Answer

9.4.2 Algorithm and complexity

Exercise 9.4.3

Outline an algorithm and its ADT(s) to solve this problem.

Hint Answer

Exercise 9.4.4

What is the worst-case complexity of your algorithm?

Hint Answer

9.4.3 Code and tests

Exercise 9.4.5

Implement and test your approach. Add tests according to your answer to the second exercise.

Hint Answer

9.5 SMS

Text messaging apps suggest completions for the word you're typing. Mobile phones usually don't have space for more than three suggestions. As there can be many words starting with the characters you typed so far, one way to reduce the suggestions is to associate a numeric score to each word and suggest only those with the highest scores. For example, a word's score can be based on how frequently it occurs in English texts.

In this problem you are going to implement two operations:

- completions: given a prefix string, return a sequence of the three highest-scoring words that start with that prefix, ordered by descending score
- initialisation: given each word and its score, in no particular order, create a collection for the first operation to use.

The completions operation returns the first three (or fewer if there aren't three) words that start with the given prefix, when ordered by descending score. When several words have the same score, it doesn't matter which get into the top three.

Here are some examples if the words and their scores are 'hello'/10, 'hi'/9, 'here'/ 5, 'there'/5 and 'hickup'/1:

Prefix	Completions
"	('hello', 'hi', 'here') or ('hello', 'hi', 'there')
'ha'	()
'he'	('hello', 'here')

Some mobile phones have less memory, some have a slower CPU. You are asked for two different approaches, to support a space–time tradeoff.

9.5.1 First approach

Exercise 9.5.1

Think of an approach that uses little memory besides the memory needed for storing all words. Outline the algorithm and ADT(s) for each operation, and explain their complexities in terms of the total number of words. Distinguish between best- and worst-case complexities if necessary.

The initialisation is done once, during the phone’s software installation, so it doesn’t have to be efficient. The completions operation should preferably take less than quadratic time.

Hint Answer

Exercise 9.5.2

Implement your approach by completing and running the following code cell. The `__init__` method includes code to read the words and scores from a file. You don’t have to understand or change that code.

The file used in the tests is in the same folder as this notebook and contains 100 common English words. You don’t have to add tests.

```
[1]: from algoesup import test

class SMS:
    """A collection of words for completing prefixes."""

    def __init__(self, filename: str) -> None:
        """Load the words and their scores from the given file.

        Preconditions: filename is the name of a text file where
        - each line is of the form 'word score'
        - scores are positive integers
        - words aren't empty nor repeated
        """

        pass # create the data structure
        with open(filename, "r") as infile:
            for line in infile:
                pair = line.split()
                word = pair[0]
                score = int(pair[1])
                pass # process the word and score
        pass # do any further processing

    def completions(self, prefix: str) -> list:
        """Return the highest-scoring words starting with prefix.
```

(continues on next page)

(continued from previous page)

Postconditions: the output is a list of at most 3 words from the file, ordered by descending scores

```
"""

```

```
pass
```

```
words_tests_100 = [
    # case,                  prefix, completions
    ('no prefix',           '',      ['the', 'of', 'and']),
    ('matches > 3',         'a',     ['and', 'as', 'at']),
    ('matches = 3',          'an',    ['and', 'an', 'any']),
    ('matches < 3',          'wi',    ['with', 'will']),
    ('matches = 0',          'z',     []),
    ('prefix = word',        'said',  ['said']),
    ('last words',          'y',     ['you', 'your']),
]

sms100 = SMS('100words.txt')
test(sms100.completions, words_tests_100)
```

Once your code passes the tests above, run the next cell, with a larger file of 10,000 English words.

```
[2]: words_tests_10000 = [
    # case,                  prefix, completions
    ('no prefix',           '',      ['the', 'of', 'and']),
    ('matches > 3',         'a',     ['and', 'as', 'at']),
    ('matches = 3',          'anx',   ['anxious', 'anxiety', 'anxiously']),
    ('matches < 3',          'tric',  ['trick', 'tricks']),
    ('matches = 0',          'glu',   []),
    ('prefix = word',        'said',  ['said']),
    ('last words',          'zo',    ['zone']),
]

sms10000 = SMS('10000words.txt')
test(sms10000.completions, words_tests_10000)
```

Hint Answer

Exercise 9.5.3

Usually the suggestions are updated after each keystroke. Your code should be able to produce three suggestions within 0.05 seconds for a typical vocabulary of 100,000 words.



Info: Jakob Nielsen states in [Powers of 10: Time Scales in User Experience](#) that 0.1 seconds is the time limit for ‘users to feel like their actions are directly causing something to happen on the screen’. To simplify, I allocate the same time for computing the suggestions and for displaying them on screen. Hence the 0.05 seconds limit.

Run the next cell. What’s the worst time you expect for 100 thousand words? Is it under the 0.05 s limit?

```
[3]: print("100 words:")
for test_case in words_tests_100:
    prefix = test_case[1]
    print('"' + prefix + '"')
    %timeit -r 5 -n 10000 sms100.completions(prefix)
print("\n10,000 words:")
for test_case in words_tests_10000:
    prefix = test_case[1]
    print('"' + prefix + '"')
    %timeit -r 5 -n 1000 sms10000.completions(prefix)
```

Hint Answer

9.5.2 Second approach

Exercise 9.5.4

Think of a different approach to solve the same problem. Aim to make the completions operation as fast as possible. You can use as much extra memory as needed. You can assume that any operation that is linear in the length of a word takes in effect constant time, because the length of commonly used words is bounded.

Again, for both operations outline the ADT(s) and algorithms, and explain their complexities.

Hint Answer

Exercise 9.5.5 (optional)

Copy the code cell with the class to below this paragraph and implement the second approach. Run the cell again, but this time with the timing code.

CHAPTER 10

TMA 01 PART 2

This study-free week is for you to work on the second part of TMA 01 and submit it together with the first part.

The rest of this chapter provides guidance for TMA 01, in addition to [Chapter 5](#).

Remember to give your TMA an overall check before you submit it:

1. Save and close your TMA notebook.
2. Have a good night's sleep.
3. Check the M269 website for any errata or clarifications about the TMA.
4. Open the TMA notebook and run all cells.
5. Read again all questions and your answers, to check if you addressed everything that is asked for, that all code tests pass, etc. Remember that [linters](#) don't flag all of your code's issues.
6. If you make any change to a code cell, restart the kernel and run all cells again.

Step 2 is crucial. You should let time pass between finishing work and checking it, to return to it with a refreshed brain and a 'new' pair of eyes to find those small silly mistakes and omissions that all of us are prone to.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

10.1 Using collections

When solving a problem about collections of items, the choice of ADT (or ADTs – you may need different ones) affects the design and efficiency of the algorithm.

When implementing a data type, determine which operations are more frequently needed and choose the data structure that supports those operations most efficiently.

Here's the list of collection ADTs, data types and data structures seen so far, with a link to the end-of-chapter summaries.

- *Chapter 4*: sequence, str, tuple, list
- *Chapter 6*: static and dynamic arrays, linked list
- *Chapter 7*: stack, (priority) queue
- *Chapter 8*: map, dict, set, set, lookup and hash tables

10.1.1 Ordered collections

There are several ADTs for collections in which the order of the elements matters.

Use this ADT	When you need to...
sequence	access any item by position
stack	process items in LIFO order, e.g. for nested structures
queue	process items in FIFO order
priority queue	process items according to a given priority

If you need to implement your own data type, the choice of data structure depends on the characteristics of your sequence: how long it can grow, which items need to be accessed, and whether there will be many insertions and removals in the middle of the sequence.

Data structure	Capacity	$\Theta(1)$ access	Shift items on insert/remove?
static array	bounded	any item	yes, except at the end
dynamic array	unbounded	any item	yes, except at the end
linked list	unbounded	start and end	no

For the same sequence of items, static arrays use less memory than dynamic arrays, which in turn use less memory than linked lists.

Python interpreters usually implement strings and tuples with static arrays, and lists with dynamic arrays.

10.1.2 Unordered collections

There are a few ADTs for collections in which the order of the elements does *not* matter.

Use a map when you need to access values by key, not by position.

Use a set when you don't want or need to consider duplicate items or you need to know

- the intersection of two collections, i.e. which (or how many) items are in both collections
- the union of two collections, i.e. which (or how many) items are in either collection
- the difference of two collections, i.e. which (or how many) items are in one collection but not in the other.

Python's `dict` and `set` data types, which are implemented with hash tables, suffice for most problems. If an unordered collection is bounded and its items are integers, consider using a lookup table.

10.2 Algorithms

General:

- Algorithms on unordered collections (maps, sets) must not rely on any particular order of the items.
- When sorting or searching, pay attention to the comparisons used.
- If you're pressed for time, try a simple algorithm that works: it may be efficient enough.
- Using bespoke algorithms instead of general-purpose ones often leads to a better solution.

Efficiency:

- The shortest code is not necessarily the most efficient.
- Avoid doing searches when possible, e.g. by looking up pre-computed values.
- Exploit space–time tradeoffs when possible: it's generally better to use more memory than to take more time.
- If you have a quadratic algorithm to solve a problem on an unsorted list, try to find an approach that first sorts the input and then applies a linear algorithm. We'll see in [Chapter 14](#) that a sequence can be sorted in better than quadratic time, so sorting followed by a linear algorithm is usually better than not sorting followed by a quadratic algorithm.

Complexity:

- The complexity of a sequence of steps is the complexity of the slowest step.

10.3 Coding style

The following is in addition to the style guide in [Chapter 5](#).

General:

- Write `set()` for the empty set and `dict()` for the empty dictionary.
- Don't chain too many methods.

Classes:

- Write class names with initial capitals and without underscores, e.g. `PriorityQueue`.
- The class name may indicate how it's implemented.
- Write a docstring for each class.

Methods:

- Write method names in lowercase, with underscores separating words, e.g. `has_item`.
- A method name should describe *what* it returns or does, not *how* it works.
- Name the first argument of a method `self`.
- Indicate the type of each parameter and of the output, with type annotations, except for the type of `self`.

- If a method's input or output is of the type of the class, write the type annotation as a string.
- Write a docstring for each method.
- A method should modify the `self` input or no input at all.
- Only a class's methods should access the instance variables.

10.4 Python

Unless a TMA 01 question explicitly states otherwise, your code can only use the types, classes, methods, functions, statements, constants and code templates introduced in Chapters 2–9 of this book. Here's a non-comprehensive index of them.

10.4.1 Language constructs

- statements: assignment (2.4), selection (3.4.4), for- and while-loops (4.3), `import` (2.1.2), `from import` (4.7), `pass` (6.3.1), `return` (2.6)
- defining functions (2.6), including docstrings and type annotations
- IPython commands `%timeit` (2.8) and `%run -i` (4.7.6)
- defining classes (6.1.3)

10.4.2 Built-in types

- operators `==` and `!=` (3.3.1)

Numbers

Types `int` and `float`:

- integer literals like `1234567`
- float literals like `5432.1` (2.1.2)
- arithmetic operators `+`, `-` (unary and binary), `*` and `/` (2.2.3)
- comparison operators `<`, `<=`, `>`, `>=` (3.3.1)
- functions `min` and `max` (2.2.3)
- operators `//`, `%`, `**` on integers only (2.2.5)

Booleans

Type `bool` (3.1.3):

- literals `True` and `False`
- operators `not`, `and`, `or`

Sequences

Type `str, range, tuple, list` ([4.9.1](#)):

- function `len`
- operators `in, +, *`
- indexing and slicing: `s[index], s[index1:index2], s[index1:], s[:index2]`

Operations on any sequence of comparable items ([4.9.1](#)):

- comparison operators `<, etc.`
- functions `min, max`
- function `sorted` with optional parameter `reverse`

Operations on lists ([4.6.4](#)):

- assignment `s[index] = new_value`
- methods `insert, append, pop`
- method `sort` (with optional parameter `reverse`) if the items are comparable

Sequence literals and constructors ([4.9.3](#)):

- strings: `'...', "...", '''...'''`, `"""...""", str(expression)`
- ranges: `range(start, end, step)`
- lists: `[item1, item2, ...], list(a_sequence)`
- tuples: `(item,), (item1, item2, ...), tuple(a_sequence)`

Dictionaries

Type `dict` ([8.2](#)):

- operators `in, not in, ==, !=`
- function `len`
- method `pop`
- index with `d[key]` and assign with `d[key] = value`
- iterate with `for key in d:` or `for (key, value) in d.items():`

Sets

Type `set` ([8.4.2](#)):

- function `len`
- methods `add, discard, union, intersection, difference`
- operators `in, not in, |, &, -` and all comparisons

10.4.3 Standard library

- functions `print` (2.4.1), `help` (2.6.1), `ord` (8.1.3), `hash`(8.3.2)
- function `floor` (2.2.4) in module `math`
- constants `pi` (2.1.2) and `inf` (6.3.1) in module `math`

10.4.4 Other

- functions `check_tests` and `test` (4.7) in module `algoesup`

10.4.5 M269 Library

The following files are in the `notebooks` subfolder of your book's folder.

Testing

- function `check` (6.2.3) in file `m269_test.py`

Ordered collections

- classes `StaticArray` (6.2.2) and `DynamicArray` (6.5.1) in file `m269_array.py`
- classes `Sequence` (6.3.1) and `ArraySequence` (6.6.2) in file `m269_sequence.py`
- class `ArrayPriorityQueue` (7.6.2) in file `m269_priority.py`
- classes `Stack` (7.1.2) and `LinkedListStack` (7.1.3) in file `m269_stack.py`
- class `Queue` (7.3.3) in file `m269_queue.py`

Image manipulation

File `m269_image.py` (4.8.6):

- functions `new_image`, `load_image`, `save_image`, `show_image`, `width`, `height`
- constants `R`, `G`, `B`, `BLACK`, `WHITE`, `SILVER`, `NAVY`, etc.

CHAPTER 11

EXHAUSTIVE SEARCH

Previous chapters covered the basic ‘ingredients’ of algorithms: sequence, selection, iteration, and the ordered and unordered ADTs and data structures used by most algorithms. This and the following chapters build on that foundation to explain the main general algorithmic techniques. New ADTs and data structures will still be introduced, as needed.

When it’s not possible to compute a solution directly from the problem instance, one general approach is to systematically generate all possible candidates, i.e. all *potential* solutions, and for each candidate, check whether it’s an *actual* solution to the problem. Such an approach is called **brute-force search** or **exhaustive search**. The candidates generated are the **search space**.

Brute-force search is a special case of a **generate and test** algorithm, which generates one object at a time and tests it for some properties. In brute-force search, each possible candidate is generated and then tested against the search criteria. In chapter 13 you will learn about binary search, another example of a generate and test algorithm, but that doesn’t generate all candidates.



Info: Other authors use generate and test as a synonym for exhaustive search, but in M269 generate and test is the general technique, with exhaustive and binary search being examples thereof.

Exhaustive search is usually a slow technique, as it generates many candidates that turn out not to be solutions. However, if we generate the candidates correctly, then it’s guaranteed to find a solution. Brute-force search can be a useful first approach to a problem, to make sure we have a correct algorithm, with appropriate tests, before we try to improve the algorithm with a different technique.

To make an exhaustive search faster, we can generate each candidate as fast as possible, test each candidate as fast as possible, or enumerate as few candidates as possible, i.e. reduce the search space. The latter has the most impact on efficiency, but we must ensure that the reduced search space still includes all solutions.

This chapter introduces examples of brute-force search and of techniques to reduce the search

space. It supports the usual learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn about cubic, factorial and exponential complexities.
- Develop and apply algorithms and data structures to solve computational problems – you will learn how to apply exhaustive search and how to make it faster.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – several algorithms in this chapter are only outlined, not fully implemented.
- Write readable, tested, documented and efficient Python code – this chapter introduces inner functions to better structure the code.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMA for what is assessed.

11.1 Linear search, again

Section 4.4 introduced linear search as ‘an algorithm that goes systematically through the sequence and checks each element’. This section provides a more general treatment.

11.1.1 Basic search

Linear search is a special case of exhaustive search in which the collection of candidates is given: generating candidates is simply iterating over the collection. Like for any generate-and-test algorithm, the test part of linear search may involve one or more candidates. For example, to find the best solution, the test involves comparing two candidates: the current one and the best candidate found so far.

At this point you may wish to skim again the algorithmic patterns in *Section 5.2* for finding all solutions, some solutions and the best solution.

Linear searches can be done on any sequence, set or map of candidates. A linear search over a stack or a priority queue destroys it, because we must remove items one by one to iterate over the collection. That’s not an issue if the input collection isn’t needed after the search. However, in general it’s best to avoid modifying the input and so we’d need to search a copy of the input.

Exercise 11.1.1

We can do a linear search over a queue without making a copy of the input collection, but not over a priority queue. Why?

Hint [Answer](#)

Exercise 11.1.2

Linear search has that name because it always has linear complexity in the worst case. True or false?

[Answer](#)

Exercise 11.1.3

Here's an algorithmic pattern for finding all solutions. It's formulated in more general terms than the corresponding one in Section 5.2.

1. let *solutions* be an empty collection
2. for each *candidate* in *candidates*:
 1. if *candidate* satisfies all search conditions:
 1. add *candidate* to *solutions*

A pattern abstracts several similar algorithms. Here, step 1 doesn't indicate the type of the output collection, step 2 doesn't specify how each candidate is obtained, step 2.1 doesn't detail the test, and step 2.1.1 doesn't say which operation is used to add a candidate to the solutions.

Make one or more steps more detailed, to obtain a pattern for linear searches that output solutions in the order they're found.

Hint Answer

Exercise 11.1.4

Here's the algorithmic pattern once more.

1. let *solutions* be an empty collection
2. for each *candidate* in *candidates*:
 1. if *candidate* satisfies all search conditions:
 1. add *candidate* to *solutions*

Modify one or more steps to get a pattern for linear searches that don't output repeated solutions. Solutions don't have to be in the order they're found.

Hint Answer

Exercise 11.1.5

For this exercise, assume the collection of candidates isn't empty. The next pattern finds *one* best solution.

1. let *best* be the first of *candidates*
2. for each *candidate* in *candidates*:
 1. if *candidate* is better than *best*:
 1. let *best* be *candidate*

Modify or add steps to obtain a pattern for a linear search that finds *all* equally best solutions. You can write '... is as good as ...' to check if two candidates are equally good. The equally best solutions may be in any order and may be duplicated, so don't specify the type of the output collection.

Hint Answer

11.1.2 Simultaneous and successive searches

You may remember the decision problem of checking if a string represents a *valid password*. It was solved with two simultaneous linear searches over the candidates (the string's characters): a search for a lowercase letter and a search for a digit. Each search uses a Boolean to record whether it was successful. When both Booleans become true, the simultaneous search stops: the string is a valid password.

Simultaneous linear searches allow us to check if the input collection, rather than an individual item, satisfies the conditions. In this example, no character can be a lowercase letter and a digit. The algorithmic pattern for simultaneous linear searches is also in [Section 5.2](#).

We can also search for each condition separately instead of simultaneously, i.e. do separate searches for a lowercase letter and for a digit. Doing one linear search for each condition is less efficient than doing a single pass over the input collection, but has other advantages. First, each linear search becomes simpler. Second, the searches can be allocated to different CPUs and executed in parallel. The time waiting for a result may not be much longer than for a single pass. Third, separate linear search functions for general conditions can be reused, making future search problems easier to solve.

If we design each linear search so that the input and output collections, i.e. *candidates* and *solutions* in the above pattern, are of the same type, then we can find the candidates that satisfy all conditions with successive separate searches, in which the solutions of one search are the candidates of the next search. This approach views searching as **filtering** the candidate collection: the conditions filter away those candidates that don't satisfy them, while the other candidates 'pass through' to the solutions collection.



Info: TM112 Block 2 Sections 2.1.3 and 2.3 introduce filtering and searching, respectively, and their connection. The algorithmic patterns in both sections are less general versions of the linear search patterns in M269.

For example, to find all white t-shirts costing less than £20 in a store, we can use three filters, i.e. three separate general linear searches: find all products of a given colour, find all products of a given kind, and find all products below a given price. Each filter takes and produces a collection of products, so they can be applied one after the other.

The order of the filters doesn't matter to obtain a correct result but we should apply them so that they **prune** the search space quickly for subsequent filters to go through as few candidates as possible. For example, a store is likely to have many more cheap products and white products than t-shirts, so searching first for t-shirts will lead to a small collection to search for white products costing less than £20.

Let's assume the filters are implemented with functions named 'colour', 'kind' and 'price'. Besides the input collection of products they have a string or integer argument to indicate which colour, kind or price to filter for. The successive filtering algorithm for the above example is:

1. let *shirts* be *kind(store, 't-shirt')*
2. let *white shirts* be *colour(shirts, 'white')*

3. let *cheap white shirts* be price(*white shirts*, 20)

Note how the output collection of a linear search is the input of the next one.



Note: The order in which you test conditions doesn't matter for correctness but may have a great impact on efficiency.

11.1.3 Sorted candidates

Consider again finding all white t-shirts under £20 in a store, using the basic linear search algorithm pattern at the start of this section, which checks each candidate against all conditions. What are the best- and worst-case scenarios? (Their complexities may be the same.)

The best case, when the linear search does the least work, is for step 2.1.1 of the pattern (adding the current candidate to the solutions) to never execute. This happens when *no* candidate satisfies the conditions, i.e. the store has no white t-shirts under £20. In the worst case, step 2.1.1 is always executed. This happens when *all* candidates satisfy the conditions, i.e. the store has nothing but white t-shirts under £20.

In both cases, the search goes through all candidates: it can't stop early, as there might be more solutions ahead. However, if the candidates are comparable, we can sort them to know when no further solutions are possible. For example, if the products are sorted by ascending price, then as soon as the current candidate costs more than £20, we can stop searching because any remaining candidates cost even more and hence won't be solutions.

Exercise 11.1.6

What are the best- and worst-case scenarios for finding all white t-shirts under £20, if the candidates (store products) are in ascending price order?

Answer

Best and worst cases rarely happen with real data, but sorting can nevertheless reduce the average run-time. We must sort and search the candidate collection so that the solutions (if there are any) appear early. For example, to find white products, we can sort products by ascending or descending colour names. If the order is ascending, then 'white' will appear towards the end of the sequence, so we must search backwards from the highest to the lowest index. If the order is descending, then white products appear towards the start of the sequence and we must search from lowest to highest index. For both orders, we're searching colours from 'z' to 'a' and stop as soon as the colour comes alphabetically before 'white', e.g. 'violet'.

Sorting can also help find the best candidate if we can sort the candidates by the criterion that is being optimised. For example, if we want to find the cheapest white t-shirt, and the products are in ascending price order, then we can stop as soon as we find a white t-shirt, as it must be the cheapest of them all. If candidates were unsorted, we'd always have to search the whole collection.

We can combine sorting and successive filtering. If we want all cheapest white t-shirts, we can first filter by kind and second by colour, then sort by price, and finally select all products that have the same price as the first one. Why is it more efficient to filter before sorting than the other way around?

We're only interested in the prices of white t-shirts, of which there are only a few compared to all products in the store. Sorting all products by price before filtering them would be a waste of time, especially if sorting takes quadratic time.

Since sorting takes longer than searching, even for the example given it's best to avoid it: a linear search for all cheapest products after filtering for white t-shirts will do the job in linear time.

However, if the same optimisation criterion is used over and over again, like finding the cheapest black shoes, the cheapest blue dresses, etc., then it's worth sorting all products by that criterion before any searches. The aim is again to spread the cost of one operation over multiple other operations. For *dynamic arrays*, the cost of copying a static array to a larger one is spread over the cost of inserting several items; here we spread the cost of sorting over the cost of linear searches.

11.2 Factorisation

This section illustrates further exhaustive search techniques with a famous and important problem in number theory and cryptography: given a positive integer n , compute all its positive integer divisors (also called factors). For example, the positive integer factors of 5 are 1 and 5, and those of 10 are 1, 2, 5 and 10.

I start by writing the problem definition and some tests in Python. To ease testing, I produce the factors in no particular order, as a set.

Function: factorisation

Inputs: n , an integer

Preconditions: $n > 0$

Output: $factors$, a set

Postconditions: the members of $factors$ are all the positive integer divisors of n

```
[1]: from algoesup import check_tests

factorisation_tests = [
    # case,           n,   factors
    ('smallest n',  1,   {1}),
    ('2 factors',   2,   {1, 2}),
    ('3 factors',   25,  {1, 5, 25}),
    ('4 factors',   10,  {1, 2, 5, 10}),
    ('5+ factors',  40,  {1, 2, 4, 10, 20, 40})
]

check_tests(factorisation_tests, [int, set])
```

OK: the test table passed the automatic checks.

To apply brute-force search I must state the problem as ‘find those candidates that satisfy these criteria’. For this problem we must find those positive integers that divide n without remainder.



Note: To apply exhaustive search, rephrase the problem at hand as a search problem.

11.2.1 Make candidates explicit

To obtain a brute-force search algorithm we must answer these questions:

- What are the candidates?
- How are they generated, one by one?
- How is each candidate tested?

The second question is very easy to answer if we have a collection of candidates to iterate over but the input is a single integer. We must first create a collection of candidates to apply linear search.

By rephrasing factorisation as a search problem, the candidates became explicit: the positive integers. Unfortunately, there are infinitely many of them, and the algorithm would never stop iterating. For brute-force search to work, two conditions must be met: the collection of candidates is finite and includes all solutions. Can you think of a finite range of positive integers that includes the factors of n ?

The smallest positive integer is 1 and no number higher than n divides n , so the integers from 1 to n include all factors of n .



Note: If the solutions are integers, determine the smallest and largest solutions to obtain a finite range of candidates.

Having a finite collection of candidates (1 to n), we turn to the other two questions of how to generate and test each one. Generating integers within a range is trivial with a for-loop. Checking if a candidate is a solution, i.e. a divisor of n , is also trivial with the modulo operation.

1. let $factors$ be the empty set
2. for each $candidate$ from 1 to n :
 1. if $n \bmod candidate = 0$:
 1. add $candidate$ to $factors$

What's the complexity of this algorithm?

The algorithm is a linear search over the integers from 1 to n and for each one executes two or three constant-time operations (just step 2.1 or also step 2.1.1). The best- and worst-case complexities are thus linear in n : $\Theta(n)$.

The code is a direct translation of the algorithm to Python.

```
[2]: from algoesup import test

def factorisation(n: int) -> set:
    """Return all positive integer divisors of n.

    Preconditions: n > 0
    """
    factors = set()
    for candidate in range(1, n + 1):
        if n % candidate == 0:
            factors.add(candidate)
    return factors

test(factorisation, factorisation_tests)

Testing factorisation...
5+ factors FAILED: {1, 2, 4, 5, 8, 40, 10, 20} instead of {1, 2, 4, 10, 20, 40, 10}
Tests finished: 4 passed (80%), 1 failed.
```

The last test fails. It's easy to see why, because the test function prints the actual and expected outputs: I forgot two factors of 40. I must replace the expected output of the last test.

```
[3]: factorisation_tests[-1] = ("5+ factors", 40, {1, 2, 4, 5, 8, 10, 20, 40})
test(factorisation, factorisation_tests)

Testing factorisation...
Tests finished: 5 passed (100%), 0 failed.
```

As explained [previously](#), an algorithm is correct if it produces an output that satisfies the postconditions for each input that satisfies the preconditions. To properly check the implementation of an algorithm, every test must associate a valid input to a valid output; otherwise the test itself is incorrect. Here, the test is incorrect because 40 satisfies the preconditions but $\{1, 2, 4, 10, 20, 40\}$ doesn't satisfy the postconditions: the set doesn't contain all positive integer divisors of 40, it's missing two of them.



Note: If the code fails a test, maybe the code is correct and the test isn't.

11.2.2 Compute solutions

Sometimes, once you find a solution, you can directly compute other solutions from it and remove them from the candidates. This reduces the remaining number of candidates to generate and test. For this problem, if we have a factor f of n , then n/f is the ‘symmetric’ factor, e.g. if 2 divides 10 then $10 / 2 = 5$ also divides 10.

The algorithm must start with an explicit collection of candidates so that the computed solutions can be removed from it.

1. let $factors$ be the empty set
2. let $candidates$ be $\{1, \dots, n\}$
3. while $candidates$ isn't empty:
 1. remove some $candidate$ from $candidates$
 2. if $n \bmod candidate = 0$:
 1. add $candidate$ to $factors$
 2. add $n / candidate$ to $factors$
 3. remove $n / candidate$ from $candidates$

This algorithm is a linear search because it iterates over a given collection, but contrary to a for-loop, it goes through the candidates in no particular order. I'll show you shortly how to implement step 3.1 in Python.

What are the best- and worst-case complexities of this algorithm? Assume step 3.1 takes constant time.

All operations within the while-loop take constant time. The number of iterations is either $n / 2$ in the best case (each iteration removes two factors) or n in the worst case (each iteration removes one factor). The algorithm is thus linear in n in the best and worst cases.

Let's move on to the code. Steps 3.2.2 and 3.2.3 of the algorithm divide integers to obtain the integer factors. In Python, the division operator `/` always produces a floating-point number, so we must use `//` (integer division) instead.

Step 3.1 is implemented with Python's `set` method `pop()`, which removes and returns an arbitrary set member.

```
[4]: def symmetric_factorisation(n: int) -> set:
    """Return all positive integer divisors of n.

    Preconditions: n > 0
    """
    factors = set()
```

(continues on next page)

(continued from previous page)

```

candidates = set(range(1, n + 1))
while len(candidates) > 0:
    candidate = candidates.pop()
    if n % candidate == 0:
        factors.add(candidate)
        factors.add(n // candidate)
        candidates.discard(n // candidate)
return factors

test(symmetric_factorisation, factorisation_tests)

Testing symmetric_factorisation...
Tests finished: 5 passed (100%), 0 failed.

```

11.2.3 Sort candidates

In [Section 11.1.3](#) you saw that sorting the products in store allowed the linear search to stop early. My first factorisation approach generates the factors in ascending order, while the second computes some factors directly. Let's try to combine both approaches and see if we can stop early due to the sorted order of the candidates.

The combined algorithm tests candidates f in ascending order ($1, 2, 3, \dots$) and, if they're a factor, adds n / f to $factors$. As f increases, n / f decreases ($n/1, n/2, n/3, \dots$), so at some point $f > n / f$. If the algorithm continues testing and increasing f from that point onwards, then it will only find the same factors that were already computed. When $f > n / f$ we have $f^2 > n$. At that point we can stop generating and testing candidates.

To sum up, by generating candidates in ascending order and by computing the symmetric factors, we can stop much earlier: when the square of the candidate is n .

1. let $factors$ be the empty set
2. let $candidate$ be 1
3. while $candidate \times candidate \leq n$:
 1. if $n \bmod candidate = 0$:
 1. add $candidate$ to $factors$
 2. add $n / candidate$ to $factors$
 2. increment $candidate$

```
[5]: def root_factorisation(n: int) -> set:
    """Return all positive integer divisors of n.

    Preconditions: n > 0
    """
    factors = set()

    # ... (rest of the function code)
```

(continues on next page)

(continued from previous page)

```

candidate = 1
while candidate * candidate <= n:
    if n % candidate == 0:
        factors.add(candidate)
        factors.add(n // candidate)
    candidate = candidate + 1
return factors

test(root_factorisation, factorisation_tests)

Testing root_factorisation...
Tests finished: 5 passed (100%), 0 failed.

```

The algorithm does \sqrt{n} iterations and each takes constant time, so the complexity is $\Theta(\sqrt{n})$. This is a vast improvement. Imagine n is one million. The original algorithm does one million iterations but this one only executes a thousand!

The new algorithm isn't just faster: it copes much better with a growing input. The larger the input, the larger the run-time reduction, compared to the full linear search from 1 to n . A picture shows it better.

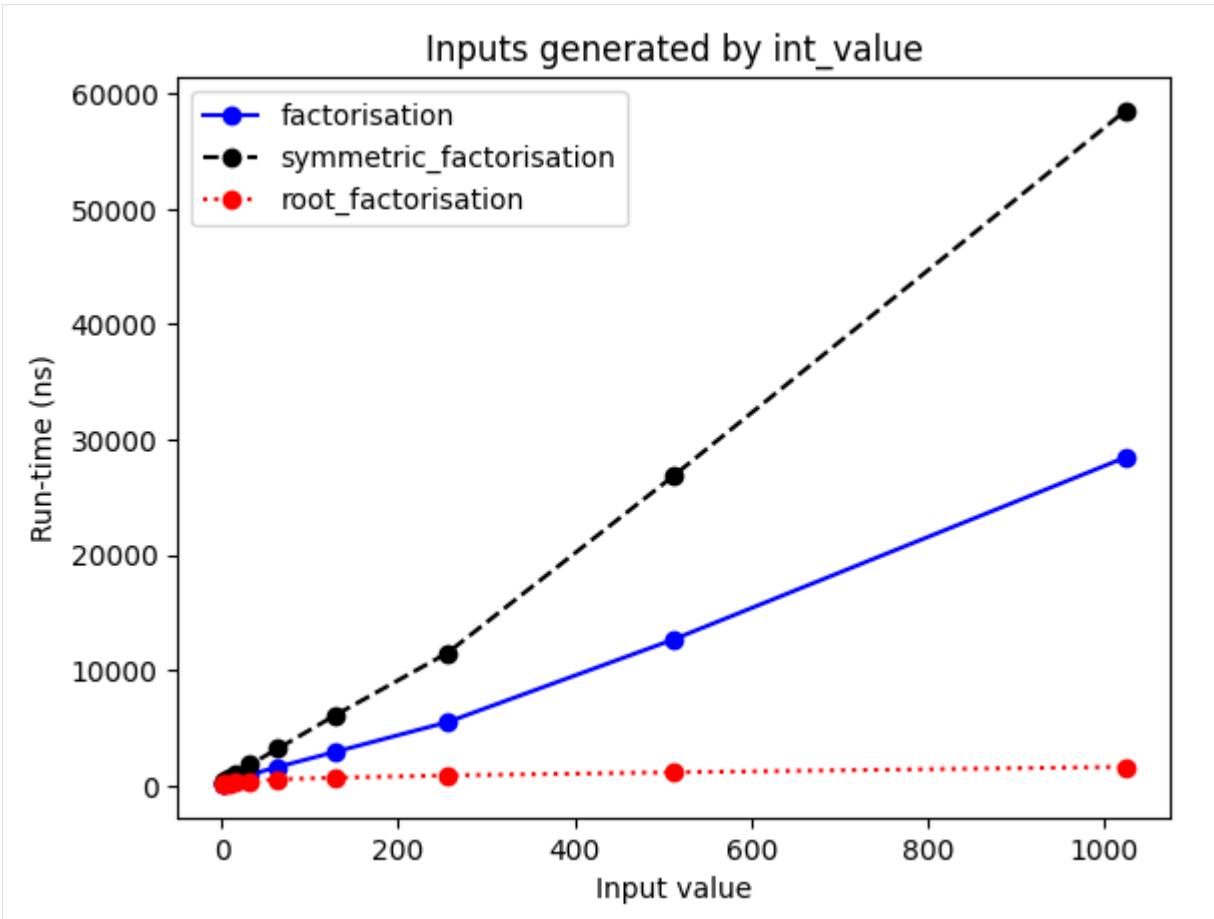
I am going to use another function of the `algoesup` module. It measures the run-times of one or more functions that take an integer as input, executing them for $n = 1, 2, 4, \dots, 1024$.

```
[6]: from algoesup import time_functions_int

time_functions_int([factorisation, symmetric_factorisation, root_
                    ↪factorisation])

Inputs generated by int_value

Input value      factorisation   symmetric_facto  root_factorisat
                152.6           275.4          150.5 ns
                184.7           282.8          151.0 ns
                237.8           422.5          217.3 ns
                328.6           592.8          217.7 ns
                576.3           1028.3         345.8 ns
                931.9           1789.3         375.6 ns
               1608.3           3208.4         528.3 ns
               2899.0           6059.1         671.1 ns
               5492.0           11496.6        863.4 ns
              12655.7           26838.6        1139.5 ns
              28435.9           58488.7        1607.7 ns
```



The basic linear search from 1 to n (Section 11.2.1) and the linear search that computes solution n / f (Section 11.2.2) have linear complexity (the run-times form straight upward lines), but the latter takes much longer, even though it makes fewer iterations, because it uses an additional set with the candidates yet to test and so the operations within each iteration take much longer to execute.



Note: Algorithms with the same complexity may have very different run-times.

The linear search up to \sqrt{n} (Section 11.2.3) seems to have constant complexity (a horizontal line) because it runs much faster than the other two and so the time growth is too small to see, but the table above the line chart shows that the run-times are not constant.

11.2.4 Prime numbers

To finish this section, here's a similar problem for practice.

A positive integer n is prime if and only if it has exactly two different positive integer factors, e.g. 1 and 4 aren't prime but 2 is. Given a positive integer n , decide whether it's prime.

Exercise 11.2.1

Is `len(factorisation(n)) == 2` an efficient way of solving the problem?

Answer

Exercise 11.2.2

Outline a more efficient algorithm to decide if n is prime.

Hint Answer

Exercise 11.2.3 (optional)

Implement your algorithm. Write a docstring and add tests.

```
[7]: import math
from algoesup import test

def is_prime(n: int) -> bool:
    pass

prime_tests = [
    # case,           n,      is_prime
    ('smallest n',  1,      False),
    ('even prime',   2,      True),
    ('n = 4',        4,      False)
]

test(is_prime, prime_tests)
```

11.3 Constraint satisfaction

Sometimes we must search for each item that satisfies the conditions. Other times we must search for multiple items that *together* satisfy the conditions. That's a form of **constraint satisfaction problem (CSP)**. The earlier example of checking if a password is valid is a CSP with two constraints: the string must include a lowercase letter and a digit. We search for two candidates (characters) that together satisfy the conditions because no character can satisfy both by itself.

CSPs occur often in business, engineering, manufacturing and many other domains. A classic CSP is timetabling: which class should be taught where and when? Constraints include teaching each class in a sufficiently large room, teaching some classes in specialised rooms (e.g. labs), and making sure no one is scheduled for different classes at the same time.

Constraints are often described with mathematical equations or inequalities. Solving a CSP amounts to assigning values to all variables so that the constraints are satisfied. There are

advanced specialised techniques to solve CSPs. In M269 we'll solve them with exhaustive search only.

This section introduces more techniques to make exhaustive search faster.

11.3.1 Problem

Consider the following CSP.

Given positive integers *sum*, *product* and *square sum*, find all distinct integers x , y and z such that $x+y+z = \text{sum}$, $x \times y \times z = \text{product}$ and $x^2 + y^2 + z^2 = \text{square sum}$.

This problem has three constraints in the form of equations and another three constraints in the form of inequalities: $x \neq y$, $y \neq z$ and $z \neq x$.

Some examples:

sum	product	square sum	x	y	z
6	6	14	1	2	3
0	6	14	-1	-2	3
21	336	149	6	7	8
33	1320	365	10	11	12

Note that this isn't a test table because the problem asks for all solutions x , y , z for each input, but I've provided only one. I actually don't know whether there are other solutions for these inputs.



Info: This is a slightly modified version of problem [Simple Equations](#).

11.3.2 Algorithm and complexity

The solutions are triples of integers, so the output will be a set of tuples (x, y, z) . We generate all possible triples with three nested loops. First we must determine a range of possible candidates for each integer.

The sum doesn't constrain the values: I can set x as small or large as I want and still find a solution ($y = -x$ and $z = \text{sum}$). However, the product equation forces each value to be in the range from $-\text{product}$ to product . If any value is outside that range, then one of the other two is a real number between -1 and 1 , not an integer.

1. let *solutions* be the empty set
2. for each x from $-\text{product}$ to product :
 1. for each y from $-\text{product}$ to product :
 1. for each z from $-\text{product}$ to product :
 1. if x , y and z satisfy the constraints:

1. add (x, y, z) to *solutions*

Steps 2, 2.1 and 2.1.1 generate the candidates and step 2.1.1.1 tests them: it checks $x \neq y \neq z \neq x$ and the three equations. A single Boolean expression for checking four constraints is too long, so I'll implement the test with a separate auxiliary function.

Each for-loop does $2 \times \text{product} + 1$ iterations: $-\text{product}, \dots, -1, 0, 1, \dots, \text{product}$. Testing a candidate requires a fixed number of arithmetic operations. The overall complexity is $(2 \times \text{product} + 1)^3 \times \Theta(1) = \Theta(\text{product}^3)$ because constants are ignored.

This is called **cubic** complexity. Quadratic algorithms are to be avoided when possible; cubic algorithms even more so.

11.3.3 Code and performance

Let's implement the algorithm to see how slowly it runs. For the Python code, I'll use the name `addition` instead of `sum` as the latter is a built-in function.

In Python we can nest functions inside each other. This is useful to write auxiliary functions that aren't used by anyone else. The inner function can access the arguments of the outer function.

```
[1]: def equations(addition: int, product: int, square_sum: int) -> set:
    """Return all triples that satisfy the constraints.

    Preconditions: addition > 0, product > 0, square_sum > 0
    Postconditions: the output has exactly all (x, y, z) such that
        - x ≠ y ≠ z ≠ x and x, y, z are integers
        - x+y+z = addition, x*y*z = product, x² + y² + z² = square_sum
    """
    def satisfies(x: int, y: int, z: int) -> bool:
        """Check if x, y and z satisfy the constraints."""
        if x == y or y == z or z == x:
            return False
        if x + y + z != addition or x * y * z != product:
            return False
        return x * x + y * y + z * z == square_sum

    solutions = set()
    for x in range(-product, product + 1):
        for y in range(-product, product + 1):
            for z in range(-product, product + 1):
                if satisfies(x, y, z):
                    solutions.add((x, y, z))
    return solutions
```

Let's test this with the simplest example.

```
[2]: equations(6, 6, 14)
```

[2]: `{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)}`

Oh dear! We get several times the same solution, only in a different order. I'll deal with that in a moment. First, let's time the execution.

[3]: `%timeit -r 3 equations(6, 6, 14)`
143 µs ± 273 ns per loop (mean ± std. dev. of 3 runs, 10,000 loops
→each)

This example ($product = 6$) required $(2 \times 6 + 1)^3 = 2197$ iterations, because each loop goes from -6 to 6 . If you divide the time shown by 2197, you will know how long it takes to generate and test each triplet x, y, z . (I don't know the run-time in advance, so you have to do the calculation yourself.)

Exercise 11.3.1

Now that you know the time it takes to generate and test each triplet, you can estimate the total run-time for any value of $product$. Such back-of-the-envelope estimates of the running time help decide if the performance is acceptable or a better algorithm is needed.

Using Python as a calculator, write an expression that computes the time, in seconds, for the algorithm to solve the earlier example with $product = 336$. If you haven't calculated the exact time for each iteration (i.e. triplet), assume each iteration takes 100 ns. (There are one billion, i.e. a thousand million, nanoseconds in a second.)

[4]: `pass`

Now write an expression to compute the time in minutes for solving the example with $product = 1320$.

[5]: `pass`

Hint Answer



Note: Knowing the complexity and the run-time for a small input, you can estimate the run-times for large inputs.

11.3.4 Don't generate equivalent candidates

The algorithm is generating all *permutations* of the same values because the order in which values are added or multiplied doesn't matter. All those solutions are equivalent; we should generate only one of them.

The easiest way to generate only one of several permutations is to sort the values, e.g. to only generate triples with $x < y < z$. This ensures the three values are distinct by construction, which makes the test simpler and more efficient, as it only has to check the equations.

1. let *solutions* be the empty set
2. for each *x* from $-product$ to *product*:
 1. for each *y* from *x* + 1 to *product*:
 1. for each *z* from *y* + 1 to *product*:
 1. if *x*, *y* and *z* satisfy the equations:
 1. add (x, y, z) to *solutions*

Imposing an order on the triple's values reduces the size of the output set and, more importantly, of the search space: steps 2.1 and 2.1.1 now generate fewer candidates for *y* and *z*.

11.3.5 Reduce the range

Another technique is to avoid generating candidates that will fail the test. In the case of integers, that means making the range of candidates as small as possible, while still making sure it contains all solutions.

We only used the product equation to set the initial search space. The equation

$$x^2 + y^2 + z^2 = \text{square sum}$$

may be useful too. Square numbers are never negative: if any of them is larger than *square sum*, then the constraint can't be satisfied. So *x*, *y* and *z* must be in the range $-\sqrt{\text{square sum}}$ to $\sqrt{\text{square sum}}$.

When we have multiple ranges, the values must be in their intersection. In this case, one range is contained in the other, because both go from $-limit$ to $+limit$ for some integer *limit*, so we take the smallest range. The algorithm becomes:

1. let *solutions* be the empty set
2. let *limit* be $\min(\text{product}, \text{floor}(\sqrt{\text{square sum}}))$
3. for each *x* from $-limit$ to *limit*:
 1. for each *y* from *x* + 1 to *limit*:
 1. for each *z* from *y* + 1 to *limit*:
 1. if *x*, *y* and *z* satisfy the equations:
 1. add (x, y, z) to *solutions*

Consider the example with *product* = 336 and *square sum* = 149. The original algorithm generates $(2 \times 336 + 1)^3 \approx 305$ million candidates; the new algorithm generates far fewer than $(2 \times \text{floor}(\sqrt{149}) + 1)^3 = 25^3 \approx 16$ thousand, because the loops for *y* and *z* never start at $-\sqrt{149}$. A vast reduction in the search space!

11.3.6 Compute part of a candidate

Last but not least, when a constraint creates a dependency between values, we can directly compute one value from the others instead of searching for it.



Note: The best way to make searches efficient is to avoid searches.

For this problem, once we have candidate values for x and y we can use one equation to determine the value of z and then check the other two equations. The sum equation is the simplest to compute z .

Here's the new algorithm, with $\text{sqrt}(\dots)$ representing the square root function, to simplify the notation.

1. let $solutions$ be the empty set
2. let $limit$ be $\min(product, \text{floor}(\text{sqrt}(square\ sum)))$
3. for each x from $-limit$ to $limit$:
 1. for each y from $x + 1$ to $limit$:
 1. let z be $sum - x - y$
 2. if $y < z$ and $x \times y \times z = product$ and $x^2 + y^2 + z^2 = square\ sum$:
 1. add (x, y, z) to $solutions$

Since z is no longer generated by a loop starting at $y + 1$, we must explicitly check for $y < z$.

Continuing with the same example, the new algorithm generates at most $25^2 = 625$ candidates, down from $25^3 \approx 16$ thousand.

11.3.7 Improved code and performance

Let's implement the final algorithm, test it and measure its performance. I won't repeat the docstring. I will use Python's square root function `sqrt`, defined in module `math`.

[6]:

```
import math

def fast_equations(addition: int, product: int, square_sum: int) ->_
    set: # noqa: D103
    solutions = set()
    limit = min(product, math.floor(math.sqrt(square_sum)))
    for x in range(-limit, limit + 1):
        for y in range(x + 1, limit + 1):
            z = addition - x - y
            if y < z and x * y * z == product and x * x + y * y + z * z == square_sum:
                solutions.add((x, y, z))
    return solutions
```

(continues on next page)

(continued from previous page)

```
solutions.add((x, y, z))
return solutions
```

[7]: `print(fast_equations(6, 6, 14))
%timeit -r 3 fast_equations(6, 6, 14)`

```
{(1, 2, 3)}
1.64 µs ± 1.83 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
```

[8]: `print(fast_equations(0, 6, 14))
%timeit -r 3 fast_equations(0, 6, 14)`

```
{(-2, -1, 3)}
1.29 µs ± 2.52 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
```

[9]: `print(fast_equations(21, 336, 149))
%timeit -r 3 fast_equations(21, 336, 149)`

```
{(6, 7, 8)}
15 µs ± 119 ns per loop (mean ± std. dev. of 3 runs, 100,000 loops each)
```

[10]: `print(fast_equations(33, 1320, 365))
%timeit -r 3 fast_equations(33, 1320, 365)`

```
{(10, 11, 12)}
35.1 µs ± 15.1 ns per loop (mean ± std. dev. of 3 runs, 10,000 loops each)
```

Now we know: each of these inputs has only one solution.

The complexity has gone down from $\Theta(\text{product}^3)$ to
 $\Theta(\min(\text{product}, \sqrt{\text{square sum}})^2) = \Theta(\min(\text{product}^2, \text{square sum}))$.

An initially cubic algorithm has become quadratic in one input or linear in another, whatever happens to be best for the particular input values. This makes a substantial difference. For $\text{product} = 1320$ and $\text{square sum} = 365$, an initially expected run-time of 30 minutes (Exercise 11.3.1) shrinks to less than a millisecond. That's the power of systematically thinking about the constraints and using them to reduce the search space.

11.4 Searching permutations

As the previous section illustrated, an exhaustive search algorithm with n nested loops can generate all permutations of n items and test which of them satisfy the search conditions. That's fine if n is small and known in advance, but if n is an input of the problem, how can we generate the permutations? This section shows how with a classic problem.

11.4.1 Problem

The **travelling salesman problem (TSP)** asks to find the shortest way to start in one place, visit other places only once and return to the start place, where ‘shortest way’ means the least total distance travelled.

A **tour** is a route that goes once through each place and returns to the start place, e.g. the route of a truck delivering or collecting goods across multiple places and returning to the warehouse. The TSP is thus asking for the tour with the least total distance, given the distances between n places. We assume there are at least two places, in order to travel to a place and back to the start place.

If instead of the distance we know the time or cost to go from one place to another, then the TSP asks for the respectively fastest or cheapest tour. If multiple tours are equally short, fast or cheap, then the output can be any of them.

Problems that ask for a collection of items that minimise or maximise some quantity are **optimisation problems**. When talking about optimisation problems in general, the quantity being minimised is traditionally called the **cost**, even if it represents something else, like distance, for particular problems. So, the TSP is an optimisation problem that asks for a tour among n places with the lowest cost, given the costs of travelling between any two places.



Info: The Robotics and AI block of TM129 introduces optimisation problems and the TSP.

To turn the informal description of the TSP into a problem definition, I must choose how to model places and costs. Whether we’re flying around the globe with a band, visiting castles in Scotland or island-hopping in the Azores, it doesn’t matter what and where the places are: all we need is the cost of travelling between any two of them. The costs must be given by numbers, so that we can add up the cost of the tour. Although costs are usually positive, I allow for zero and negative numbers, e.g. if we wish to represent the cost below or above a given travel budget.

There are several ways of representing the costs between pairs of places. A simple way of quickly looking up costs is to store them in a matrix. This means that places must be represented by natural numbers, in order to index the matrix. We have to decide whether the cost of going from *origin* to *destination* is in row *origin* and column *destination*, or row *destination* and column *origin*. It doesn’t matter which way it is, as long as we state it in the preconditions, so that the user knows how to fill the input matrix.

Next, the output: the best tour. It can be represented by a sequence of integers, indicating in which order the places are visited. The postconditions must state that all places occur in the tour and that the first and last place are the same. They must also define the optimisation criterion: the tour’s cost.

Function: TSP

Inputs: $costs$, a $n \times n$ matrix of numbers

Preconditions: $n \geq 2$; $costs[i][j]$ is the cost of travelling from i to j

Output: $tour$, a sequence of integers

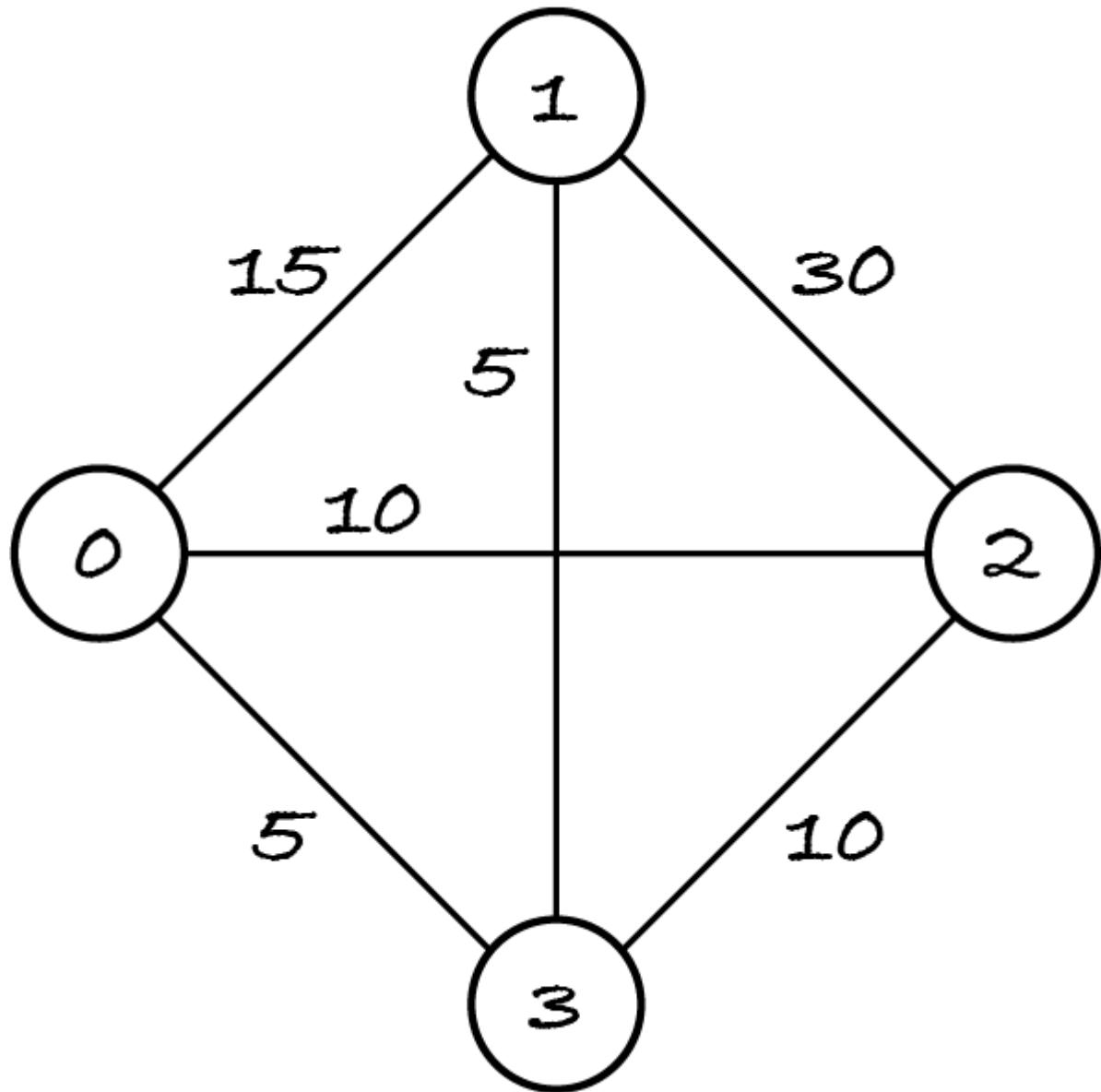
Postconditions:

- tour has length $n + 1$, with $\text{tour}[0] = \text{tour}[n]$
- every integer from 0 to $n - 1$ occurs in tour
- $\text{costs}[\text{tour}[0]][\text{tour}[1]] + \text{costs}[\text{tour}[1]][\text{tour}[2]] + \dots + \text{costs}[\text{tour}[n - 1]][\text{tour}[n]]$ has the lowest possible value

Because the tour has length $n + 1$, with one duplicate place, all other places must occur once in the tour.

Here's a depiction of the example I'll use, with $n = 4$. In this example, travelling from A to B costs the same as from B to A. The places are numbered from 0 to 3 and indicated by circles. The cost of travelling between two places is indicated next to the line connecting the two places.

Figure 11.4.1



Can you find at least two tours with the same lowest cost? Two tours are different if their start/end place is different or if they visit the places in a different order.

Some of the tours with lowest cost 40 are $(0, 1, 3, 2, 0)$, $(1, 3, 2, 0, 1)$ and $(3, 1, 0, 2, 3)$.

11.4.2 Algorithm

The exhaustive search algorithm is like a linear search for the best candidate, where the candidates are all possible tours and being better means to have a lower cost.

Tours that go through the same places in the same order have the same cost, so to find the best tour it doesn't matter which place we start from, as the example above shows. We can choose place 0 to be the start and end place and generate all permutations of $(1, \dots, n - 1)$ for the intermediate places.

We compute the cost of each candidate tour and see if it improves the current best cost, which is initially infinite. (If we were maximising a quantity, we'd initialise it with negative infinity.)

1. let *best cost* be infinite
2. for each *places* that is a permutation of $(1, 2, \dots, n - 1)$:
 1. let *this tour* be *places* with 0 prepended and appended
 2. let *cost* be $\text{costs}[\text{this tour}[0]][\text{this tour}[1]] + \dots + \text{costs}[\text{this tour}[n - 1]][\text{this tour}[n]]$
 3. if $\text{cost} < \text{best cost}$:
 1. let *best cost* be *cost*
 2. let *tour* be *this tour*

Exercise 11.4.1

In many real-life situations, the cost of going from A to B is the same as the cost of going from B to A, like in the example above. In such cases, will the algorithm generate different candidate tours with the same total cost?

Hint Answer

11.4.3 Complexity

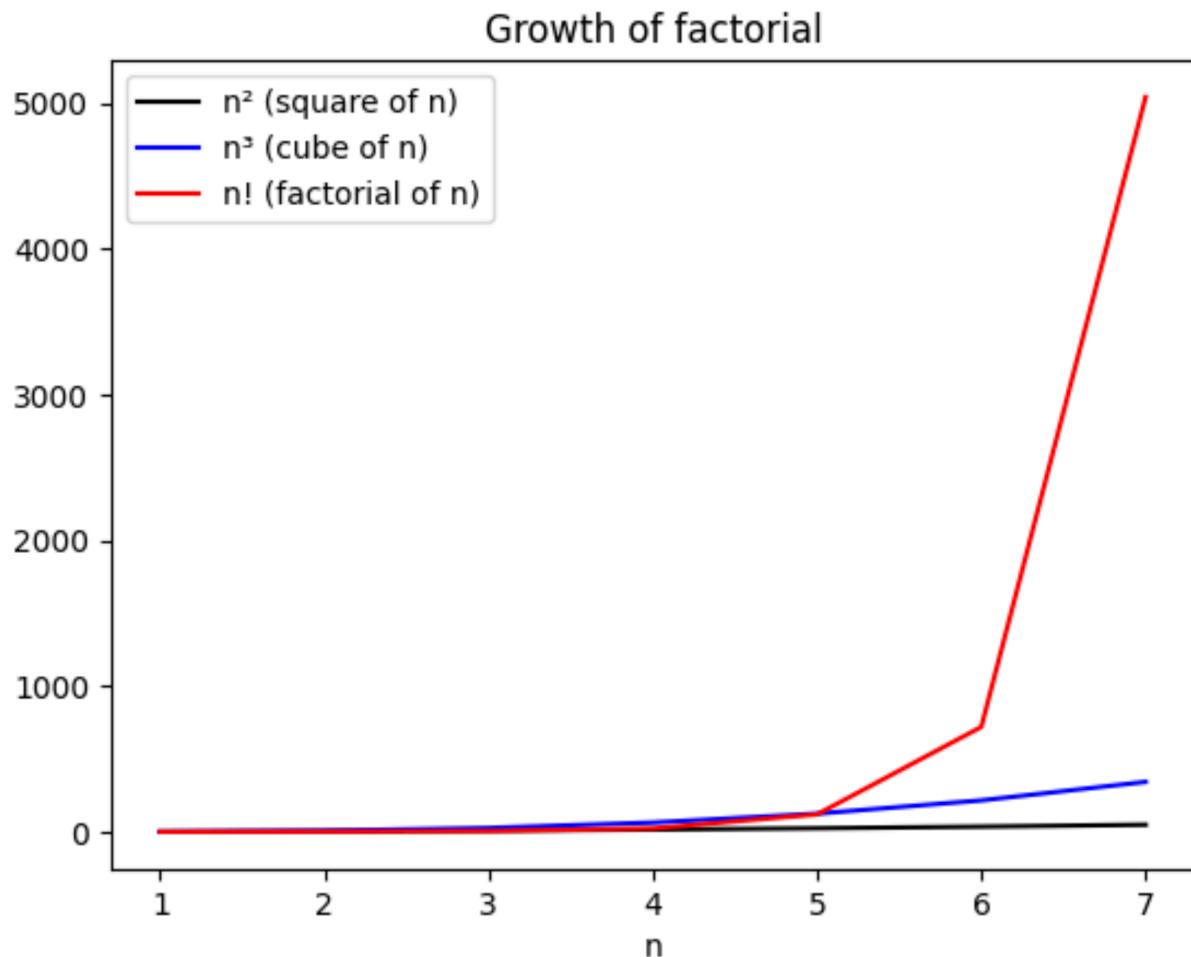
There are $n \times (n-1) \times \dots \times 2 \times 1$ permutations of n items, because the first item in the permutation can be any of the n items, the second item in the permutation can be any of the remaining $n-1$ items, and so on, until only one item remains to be put in the last position.

The product of all integers from 1 to n is written $n!$ and is called the **factorial** of n . There's only one permutation of zero items, the empty sequence, so $0! = 1$.

The exhaustive search for the best tour among n places only generates the permutations of $(1, \dots, n - 1)$, because the start/end place is fixed, so the complexity is $\Theta((n-1)!)$.

The factorial grows much faster than the square or the cube of a number. A picture shows it best.

Figure 11.4.2



As the chart shows, $5!$ is about the same as 5^3 (the red and blue lines cross at $n = 5$) but from then on the factorial function leaves the cubic function in the dust, and n^2 looks like a constant function (the black line is flat) in comparison to $n!$.

I once asked the driver delivering my supermarket online shopping how many customers he served in one tour with a full van. He said about fifteen. Assuming we can compute the cost of each tour in $1 \mu\text{s}$, we can obtain a van's best tour in $15! \mu\text{s}$. Python's `math` module provides a function to compute the factorial.

```
[1]: from math import factorial

print(factorial(15), "μs")
1307674368000 μs
```

That's a huge number. Let's see how many days that is.

```
[2]: MS_PER_DAY = 24 * 60 * 60 * 1000 * 1000 # microseconds in a day
print(factorial(15) // MS_PER_DAY, "days")
15 days
```

By comparison, if the van delivers to 10 customers, then the best tour can be computed in just ...

```
[3]: MS_PER_S = 1000**2 # microseconds in a second
      print(factorial(10) // MS_PER_S, "seconds")
      3 seconds
```

This shows again how fast the factorial function grows.

As you may imagine, delivery companies don't use exhaustive search, with factorial complexity, to compute the best tour. They use **heuristic** algorithms which compute an approximate 'good enough' solution, rather than the optimal one. A heuristic algorithm for the TSP will be presented in *Exercise 18.3.1*.

Exercise 11.4.2

The previous exercise showed that the algorithm generates 'symmetric' tours. Would the complexity improve if the algorithm could be modified to not generate them?

Answer

11.4.4 Code

To generate permutations, we use function `permutations` in module `itertools`. The function is meant to be used in a for-loop because it returns the permutations one by one, as a tuple. The function's argument is an iterable collection of items.

```
[4]: from itertools import permutations

for permutation in permutations(["travelling", "salesman", "problem"]):
    print(permutation)

('travelling', 'problem', 'salesman')
('travelling', 'salesman', 'problem')
('problem', 'travelling', 'salesman')
('problem', 'salesman', 'travelling')
('salesman', 'travelling', 'problem')
('salesman', 'problem', 'travelling')
```

Let's construct the matrix of the example input. The cost of 'travelling' from a place to itself can be any number, because it won't be used by the algorithm, but it's usually set to zero.

```
[5]: four_places = [
    [0, 15, 10, 5],      # cost from 0 to other places
    [15, 0, 30, 5],      # cost from 1 to other places
    [10, 30, 0, 10],     # cost from 2 to other places
    [5, 5, 10, 0],       # cost from 3 to other places
]
```

I use slightly different variable names from the algorithm to keep the code lines short.

```
[6]: from itertools import permutations
import math

def tsp(costs: list) -> tuple:
    """Solve the travelling salesman problem.

    Input: costs is a n*n matrix of numbers
    Preconditions:
        - n > 1
        - costs[i][j] is the cost of travelling from i to j
    Postconditions:
        - len(output) = n + 1
        - output[0] = output[n]
        - every integer from 0 to n - 1 occurs in the output
        - costs[output[0]][output[1]] + ... + costs[output[n-1]][output[n]]
            has the lowest possible value
    """
    best_cost = math.inf # positive infinity (Section 6.8)
    n = len(costs)
    # generate all permutations of (1, ..., n-1)
    for places in permutations(range(1, n)):
        # tuple literals with 1 item need extra comma (Section 4.5)
        tour = (0,) + places + (0,)
        cost = 0
        for index in range(0, n):
            cost = cost + costs[tour[index]][tour[index + 1]]
        if cost < best_cost:
            best_cost = cost
            best_tour = tour
    return best_tour

tsp(four_places)
```

[6]: (0, 1, 3, 2, 0)

There are two equally best tours starting and ending at place 0, namely (0, 1, 3, 2, 0) and (0, 2, 3, 1, 0), but only the one generated first is output as the second one doesn't improve the best cost.

11.5 Searching subsets

For the TSP, the order in which places are visited impacts the total cost and so the output must be a sequence. For other problems, the output is a set and we must generate subsets instead of permutations of a collection of items. Here's one such problem.

11.5.1 Problem

Many products (cars, dishwashers, etc.) are designed and manufactured as product lines with a set of configurable features, to reduce costs. Not all features are compatible with each other, e.g. the features ‘petrol engine’ and ‘diesel engine’ are mutually incompatible.

Given the features and the pairs of incompatible features, how many different products can be made? This is another example of a CSP: the constraints are given as incompatible feature pairs.



Info: This is an adaptation of problem Geppetto.

Here’s one possible definition of the problem. How each feature is represented (an integer, a string, etc.) is irrelevant.

Function: feasible products

Inputs: *features*, a set of objects; *incompatible*, a set of pairs of objects

Preconditions:

- *features* isn’t empty
- every pair in *incompatible* consists of two different objects in *features*
- if *incompatible* has feature pair (A, B), then it hasn’t pair (B, A)

Output: *products*, an integer

Postconditions: *products* is the number of non-empty subsets of *features* that don’t contain a pair of objects in *incompatible*

The second precondition states that no feature is incompatible with itself. The third precondition prevents redundant incompatibility information.

Let’s think about some tests. What are the edge cases?

The smallest possible input is a single feature which therefore has no incompatibilities. Other edges cases (with more than one feature) are no incompatibilities and all features are mutually incompatible.

To keep lines of code short, I represent features with integers instead of strings. I use tuples to represent pairs.

```
[1]: from algoesup import check_tests

feasible_products_tests = [
    # case,           features,      incompatible, products
    ('smallest input', {1},        {},                  1),
    ('all compatible', {1, 2},     {},                  3),
    ('no compatible', {1, 2},     {(1, 2)},            2),
]
```

(continues on next page)

(continued from previous page)

```

('some compatible', {1, 2, 3, 4}, {(1, 2), (3, 4)}, 8)
]

check_tests(feasible_products_tests, [set, set, int])

Error: in test "smallest input", {} must have type set.
Error: in test "all compatible", {} must have type set.

```

The ‘all compatible’ test has output 3 because there are two products with one feature each and one product with both features. The latter isn’t a feasible product when both features are incompatible, so there are only two single-feature products for the ‘no compatible’ test.

Can you explain why there are eight feasible products for the last problem instance?

There are four single-feature products and another four products with two features: one from the first incompatible pair and another from the second pair. Those four products have features 1 and 3, 1 and 4, 2 and 3, and 2 and 4. There are no products with three or four features as they would include two incompatible features.

11.5.2 Algorithm

The problem can be solved with an exhaustive search: generate each non-empty subset of features and test whether it includes a pair of incompatible features. If not, we found a subset of compatible features and can increment the product counter. To check if a candidate feature subset is a solution, we make a linear search over all pairs of incompatible features and test if both are in the current candidate.

The overall algorithm thus consists of an exhaustive search within an exhaustive search.

In mathematics, a **k-combination** is a selection of k items from a collection, without considering the order in which they were selected. If the collection from which items are selected is a set, then a k -combination is a subset of size k . The 0-combination is the empty set.

For this problem, k is the number of features to be put in the product. The algorithm has to generate and test all k -combinations, for each k from 1 to the total number of features.

1. let $products$ be 0
2. for each k from 1 to $|features|$:
 1. for each $product$ that is a k -combination of $features$:
 1. if $\text{feasible}(product, incompatible)$:
 1. let $products$ be $products + 1$

Step 2.1.1 tests the current product candidate with an auxiliary Boolean function that does the linear search.

Exercise 11.5.1

Copy the above algorithm and change it so that instead of the number of feasible products it computes a largest set of mutually compatible features. (This is known as the **maximal independent set** problem.) The output variable should be called *compatible*.

Answer

11.5.3 Complexity

When generating a subset, we have two choices for each item: either we put it in the subset or we don't. This means there are 2^n subsets of a set with n items.

A simple rule of thumb is that $2^{10m} = (2^{10})^m = 1024^m$ is about 1000^m , so sets of 10, 20 and 30 items (m is 1, 2, and 3) have about a thousand, a million and a billion subsets, respectively.

The algorithm does a linear search over the incompatible pairs for each subset. The worst-case complexity is therefore $\Theta(2^n \times |\text{incompatible}|)$, with $n = |\text{features}|$. The complexity is said to be **exponential** when it's of the form c^n , with c some constant greater than one and n the size of one of the inputs.



Info: MU123 Unit 13 and MST124 Unit 3 Section 4 introduce exponential functions.

Exponential algorithms with $c = 2$ become slow very fast, but not as fast as factorial algorithms, which take eons to finish even for very small inputs, assuming the hardware would last that long. Here's a comparison of several functions.

n	n^2	n^3	2^n	$n!$
0	0	0	1	1
5	25	125	32	120
10	100	1,000	1,024	3,628,800
15	225	3,375	32,768	1,307,674,368,000
20	400	8,000	1,048,576	2,432,902,008,176,640,000
25	625	15,625	33,554,432	15,511,210,043,330,985,984,000,000

For example, if generating and testing one subset of features takes 1 ms, then the exponential algorithm takes about 33.5 thousand seconds (that's over 9 hours!) for 25 features, a rather small input value.

If generating one tour and computing its cost also takes 1 ms, then the factorial exhaustive search algorithm for the TSP in the previous section takes

```
[2]: from math import factorial

MS_PER_YEAR = 365 * 24 * 60 * 60 * 1000 # milliseconds in a year
print(factorial(25) // MS_PER_YEAR // 1000**3, "billion years")
```

491857 billion years

to find the best tour to visit 25 places and return to the start place.



Note: Algorithms with best-case exponential complexity can only be applied to very small input values. Algorithms with best-case factorial complexity are practically useless.

11.5.4 Code

To generate subsets, we'll use another function from the `itertools` module: `combinations`. It takes a collection of items and an integer k , and generates one by one all k -combinations of those items. Each combination is represented with a tuple although conceptually it's a set. Here's a simple example.

```
[3]: from itertools import combinations

items = {"some", "words"}
for size in range(len(items) + 1):
    for subset in combinations(items, size):
        print(subset)

()
('some',)
('words',)
('some', 'words')
```

Let's implement the algorithm above.

```
[4]: from itertools import combinations
from algoesup import test


def feasible_products(features: set, incompatible: set) -> int:
    """Return the number of subsets of features without
    incompatibilities.

    Preconditions:
    - len(features) > 0
    - incompatible is a set of pairs of distinct elements of features
    - if pair (a, b) is in incompatible, pair (b, a) isn't
    """
    def feasible(product: tuple) -> bool:
        """Check if product hasn't two incompatible features."""
        for pair in incompatible:
            if pair == product or pair == (product[1], product[0]):
                return False
        return True
    return sum(feasible(subset) for subset in combinations(features, len(features)))
```

(continues on next page)

(continued from previous page)

```
if pair[0] in product and pair[1] in product:  
    return False  
return True  
  
products = 0  
for size in range(1, len(features) + 1):  
    for product in combinations(features, size):  
        if feasible(product):  
            products = products + 1  
return products  
  
test(feasible_products, feasible_products_tests)  
Testing feasible_products...  
Tests finished: 4 passed (100%), 0 failed.
```

Exercise 11.5.2

The **0/1 knapsack** problem, another classic optimisation problem, goes as follows. Given a set of items as weight–value pairs, and given the largest weight a rucksack can carry without bursting, find the highest-valued subset of items that can be packed.

The name of the problem comes from the fact that the solution has 0 or 1 of each item.

Outline an algorithm.

Hint Answer

11.6 Practice

The following problems can all be solved with some form of exhaustive search. They are in increasing difficulty order, in my opinion.

None of the exercises asks for code, but if you have the time, you should try to implement and test at least one of the algorithms.

11.6.1 Duplicates

Given a sequence of items, decide if there are any duplicates, e.g. (1, ‘hi’, 3, 1, 4) has a duplicate but (4, ‘hi’, true) hasn’t.

Exercise 11.6.1

The Python expression `len(set(items)) < len(items)` does the job for a list of hashable items. Explain why.

Answer

Exercise 11.6.2

The previous approach uses extra memory for the set and, in Python, only works for hashable items. Write an exhaustive search algorithm that takes as input a sequence *items* and outputs a Boolean *duplicates*. The algorithm shouldn't use any additional data structures and can only compare items using the (in)equality operation.

Hint Answer

Exercise 11.6.3

What are the best- and worst-case complexities of your approach?

Answer

Exercise 11.6.4

Now assume that items are comparable with $<$, $>$, \leq , etc. Outline a different exhaustive search approach.

Hint Answer

11.6.2 Subset sum

In an *earlier problem* you were asked to find all pairs of products that together cost exactly a voucher's amount. Let's now consider a more general formulation of that problem:

Given a sequence of products and the amount of a voucher, find a subset of products so that their total price comes as close as possible to the voucher amount without exceeding it.

Exercise 11.6.5

What kind of problem is this?

Answer

Exercise 11.6.6

Outline a simple algorithm to solve the problem. You don't need to think about reducing the search space.

Hint Answer

Exercise 11.6.7

Can you think of a way of reducing the search space or stopping early to make the exhaustive search more efficient?

Hint Answer

11.6.3 Substring

Imagine Python's `in` operator didn't apply to strings. How could you determine if string s is a substring of string t ? You can assume that s isn't longer than t ; otherwise s couldn't be a substring.

Exercise 11.6.8

Outline an algorithm to solve the problem. You can use any *string operation* you have learned, except the substring operation, of course.

Hint Answer

Exercise 11.6.9

What are the worst-case scenario and complexity of your algorithm?

Hint Answer

11.6.4 PIN

Bob forgot his 4-digit PIN. He remembers that all digits are different and that the PIN is divisible by each digit. Find all such PINs to jog Bob's memory.

Exercise 11.6.10

What kind of problem is this?

Answer

Exercise 11.6.11

Outline two different approaches to solve the problem.

Hint Answer

Exercise 11.6.12

What is the worst-case complexity of each approach?

Hint Answer

11.7 Summary

A **generate-and-test** algorithm generates one object at a time and checks if it has some properties. The properties may involve other objects, e.g. to check if the generated object is better than the best found so far.

An **exhaustive** (or **brute-force**) search is a generate-and-test algorithm that generates possible candidates and tests if each one satisfies the search criteria. The candidates generated are the **search space**.

A **linear search** is an exhaustive search over a given collection of candidates. The generation part simply iterates over the collection. If there's no explicit input collection of candidates, then one must be defined in order to apply linear search. For example, solving the factorisation problem requires thinking about the range of possible factors.

Searching for all items that satisfy some criteria filters the input collection. A conjunction of criteria can be implemented as a succession of filters, each filter searching the output collection of the previous filter for matches against one criterion.

11.7.1 Problems

Exhaustive search algorithms can be used to solve decision, optimisation and constraint satisfaction problems.

A decision problem on a collection can sometimes be solved with a search, either for several items that together satisfy the criteria, or for one item that doesn't satisfy the criteria. For example, we decided if a string is a valid password by searching for a lowercase letter and a digit, and in [Section 4.8.1](#) we decided if a string models a DNA strand by searching for a character different from A, C, G and T.

A **constraint satisfaction problem (CSP)** asks to find two or more items that together satisfy some criteria, so each candidate is a sequence or subset of items. For example, finding all suitable PINs involves searching for all 4-digit sequences that satisfy some properties.

An **optimisation problem** asks for the sequence or subset of items that minimises or maximises some quantity. Optimisation problems often have constraints too. Three classic optimisation problems are:

- the **travelling salesman problem (TSP)** asks for a sequence of places that are constrained to be a tour and minimise the travel cost (or time or distance)
- the **0/1 knapsack problem** asks for a maximum-value subset of items that satisfy the constraint of fitting in the knapsack
- the **maximal independent set problem** asks for a largest subset of items that satisfy the constraint of being mutually compatible.

Optimisation problems are in practice often solved by **heuristic** algorithms that are efficient but compute an approximate solution.

11.7.2 Complexities

An algorithm has **cubic complexity** if it is $\Theta(n^3)$.

The complexity $\Theta(c^n)$ is said to be **exponential** if n is the size of the input and $c > 1$. There are 2^n subsets of a set of n items, so generating them all takes exponential time.

There are $n! = 1 \times 2 \times \dots \times (n-1) \times n$ permutations of n items, so generating all of them takes **factorial** time.

Algorithms with cubic complexity are much more efficient than those with exponential complexity, which in turn are much more efficient than those with factorial complexity.

11.7.3 Reducing the search space

Reducing the search space is the most effective way of improving the run-time of brute-force search. Techniques to reduce the search space include:

- avoiding generating ‘symmetric’ candidates
- computing directly further solutions when one is found and removing them from the candidates
- sorting candidates to put solutions early on and to stop when no further solution will appear.

11.7.4 Python

Functions `sqrt` and `factorial` in module `math` compute the square root of a number and the factorial of a non-negative integer, respectively.

Method `pop` on a set removes and returns a random element of the set.

To generate all permutations of a collection of items, write

```
from itertools import permutations

for permutation in permutations(items):
    # do something with the permutation tuple
```

To generate all subsets of a collection of items, write

```
from itertools import combinations

for size in range(len(items) + 1):
    for subset in combinations(items, size):
        # do something with the subset tuple
```

You can nest function definitions within function definitions. The **inner function** can access the arguments of the outer function.

CHAPTER 12

RECUSION

This chapter introduces **recursion**, a programming technique in which a function calls itself. It has been proven that every computable problem can be solved with either an iterative algorithm that uses loops, or a recursive algorithm that doesn't. You can see recursion as a style that can be applied to various algorithmic techniques. For some problems, recursive algorithms are shorter and simpler than iterative ones. It's useful to know how to program recursively.

There's a group of programming languages, called functional languages, where recursion is the main or only means of iterating, instead of using for- and while-loops. Such languages have specialised but important applications in telecommunications, finance and other domains. So another reason to have recursion in your algorithmic toolbox is to be able to use functional languages.



Info: If you're interested, the Wikipedia article on [functional programming](#) provides a good overview and points to industrial applications.

For pedagogical reasons, I'll introduce recursion with simple examples for which it's not the best choice in Python, before moving on to more appropriate examples. Complexity analysis of recursive algorithms is explained in the next chapter.

This chapter supports these learning outcomes:

- Develop and apply algorithms and data structures to solve computational problems – you will learn how to design recursive algorithms.
- Write readable, tested, documented and efficient Python code – you will learn how to write recursive functions in Python.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

12.1 The factorial function

The *factorial function* is a classic simple example to introduce recursion.

12.1.1 Definition and algorithm

The factorial function can be defined like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 \times 2 \times \cdots \times (n-1) \times n & \text{if } n > 0 \end{cases}$$



Info: MST124 Unit 3 Section 1.3 calls this a piecewise-defined function.

The product of 1 to n is the product of 1 to $n-1$, multiplied by n . The product up to $n-1$ happens to be the factorial of $n-1$, so the definition can be rewritten as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

This is a **recursive** definition: the factorial of n is defined in terms of the factorial of $n-1$. Eventually n is small enough for the factorial to be directly computed. The case $n = 0$ is called the **base case**. The case $n > 0$ is defined with a **recurrence relation** that shows how the value for n is computed from the value for $n-1$.



Info: MST124 Unit 10 Section 1.3 introduces recurrence relations.

The mathematical notation is cumbersome to typeset, so in M269 we write recursive definitions with one bullet point per case and with the condition before the expression:

- if $n = 0$: $n! = 1$
- if $n > 0$: $n! = (n-1)! \times n$

A recursive definition must satisfy three conditions:

- The cases must cover all possible input values. Otherwise, the output wouldn't be defined for at least one input value. Here, the two cases together cover all natural numbers.
- The lowest possible input value must be a base case because it can't be further decreased. Here, the lowest input value is zero. You'll see examples with more than one base case.
- The recurrence relation must decrease the input to eventually reach the base case(s). In the next chapter you'll see recursive definitions that decrease by more than one.

We can restate the definition in our usual notation:

Function: factorial

Inputs: n , an integer

Preconditions: $n \geq 0$

Output: $product$, an integer

Postconditions: $product = 1$ if $n = 0$, otherwise $product = \text{factorial}(n-1) \times n$

The algorithm just follows the definition.

1. if $n = 0$:

1. let $product$ be 1

2. otherwise:

1. let $product$ be $\text{factorial}(n-1) \times n$

The last step has a **recursive call**: the algorithm calls the same function, but with a smaller argument; otherwise the algorithm would never stop. There's nothing magical or mysterious about a recursive call. It's a function call like any other: the arguments are passed to the function, which returns a value after doing its job. The adjective 'recursive' just indicates that the function being called happens to be the same as the function doing the call.

12.1.2 Code

The beauty of recursion is that once the problem is recursively defined, the algorithm and the code follow that definition.

```
[1]: def factorial(n: int) -> int:
    """Return the factorial of n.

    Preconditions: n >= 0
    Postconditions:
        - if n = 0, then the output is 1
        - otherwise the output is 1*2*...*(n-1)*n.
    """
    if n == 0:
        return 1
    else:
        return factorial(n - 1) * n

factorial(5)
```

```
[1]: 120
```

Let's add some print statements to see what each recursive call is computing. Each call indents the message by four spaces.

```
[2]: def factorial_printed(n: int, level: int) -> int:
    """Return the factorial of n.
```

(continues on next page)

(continued from previous page)

```
Preconditions: n >= 0, level >= 0
Postconditions: the output is 1 for n=0, otherwise 1*...*(n-1)*n
"""

print("    " * level, "factorial of", n, "=")
if n == 0:
    print("    " * level, 1)
    return 1
else:
    product = factorial_printed(n - 1, level + 1) * n
    print("    " * level, "*", n, "=", product)
    return product

factorial_printed(3, 0)

factorial of 3 =
    factorial of 2 =
        factorial of 1 =
            factorial of 0 =
                1
                *
                1 = 1
            *
            2 = 2
        *
        3 = 6
[2]: 6
```

This shows, for example, that the factorial of 3 computes the factorial of 2 and, once the result is obtained, multiplies it by 3. The algorithm does indeed multiply all integers from 1 to n .

The way that the indentation level increases and then decreases shows that this and many other recursive algorithms have two phases. The first phase proceeds top-down from the given problem instance and makes it progressively smaller until the answer can be directly computed, in this example for $n = 0$. The second phase proceeds bottom up from the base case. It extends the answers to increasingly large subproblems ($0!$, $1!$, $2!$, etc.) until it obtains the answer to the original problem instance.

12.1.3 The call stack

When function A calls function B, A is on hold until B returns. To be able to resume A where it left off to call B, the interpreter must remember the value of each variable in A. This information is stored before calling B and restored after B returns. The *last* function called is the function currently running, so it's the *first* function to return. This *LIFO* relation allows the interpreter to store the information in a stack, named the **call stack**. Each item on the call stack is called a **stack frame**, which includes the function's variables (both arguments and local variables), the instruction where the function resumes execution after calling another function, and any other data the interpreter needs.

Here's a [visualisation](#) of call stacks in general, then for the factorial function. The visualisation

was produced for the previous edition of M269: the coding style is different and it mentions a Unit 3 that no longer exists.

Interpreters reserve a limited amount of memory for call stacks. If many functions are waiting for others to complete, then the stack may get full, which leads to an error. Limiting the number of calls is good to detect infinite calls, e.g. if A calls B which calls C which then calls A again and so on, or if a recursive call doesn't decrease the input. Unfortunately, limiting the number of calls also leads to errors for correct recursive functions, if the input is too large.

```
[3]: factorial(3000)
-----
←-----  
RecursionError                                         Traceback (most recent...  
    ↪call last)  
Cell In[3], line 1  
----> 1 factorial(3000)  
  
Cell In[1], line 12, in factorial(n)  
    10     return 1  
    11 else:  
---> 12     return factorial(n - 1) * n  
  
Cell In[1], line 12, in factorial(n)  
    10     return 1  
    11 else:  
---> 12     return factorial(n - 1) * n  
  
    [... skipping similar frames: factorial at line 12 (2974 times)]  
  
Cell In[1], line 12, in factorial(n)  
    10     return 1  
    11 else:  
---> 12     return factorial(n - 1) * n  
  
RecursionError: maximum recursion depth exceeded
```

Interpreters usually allow users to change the limit, but Python's default suffices for our needs in M269.



Note: In Python, recursive functions that decrease the input by one can't take large inputs.

12.2 Recursion on integers

Let's step back from the factorial example and see how to design a recursive algorithm on integers. Consider a problem on an integer n with precondition $n \geq \text{lowest}$. Computing the factorial is such a problem, with $\text{lowest} = 0$.

12.2.1 Algorithm

A recursive algorithm repeatedly decreases n by one each time, until it can't be further decreased, i.e. until the input is lowest . At that point we reached the base case and the answer is directly computed. Then, the output for lowest is used to obtain the answer for $\text{lowest} + 1$. That output is then used to obtain the output for $\text{lowest} + 2$ and so on, until we get the output for the original input n .

The algorithmic pattern for a recursive decrease by one function $f(n)$ is:

1. if $n = \text{lowest}$:
 1. let *solution* be ...
2. otherwise:
 1. let *subproblem* be $n - 1$
 2. let *subsolution* be $f(\text{subproblem})$
 3. let *solution* be based on *subsolution* and n

The algorithm first checks for the base case (step 1) and computes the output directly (step 1.1). Otherwise, the algorithm decreases the input (step 2.1), makes a recursive call to solve the smaller subproblem (step 2.2), and finally combines the ‘subsolution’ (the solution for the subproblem) with n to get the solution for the original problem (step 2.3). As you shall see, for some problems the solution can be computed from the subsolution only, without using the value of n , but that's rare.

To sum up, a recursive algorithm usually consists of four steps:

- check and compute the base case
- decrease the input
- recur
- combine.

If the last three steps are simple and short, they are merged into a single expression, as done for the factorial.

The second step usually decreases the input by one to make sure it reaches the base case. For example, if $\text{lowest} = 3$ and the input were decreased by two at a time, then the recursion wouldn't stop for any even n , because the successive inputs would be ..., 4, 2, 0, -2, -4, ...

12.2.2 Recursive definition

To fill in the algorithmic pattern, you only need to come up with the recursive definition and for that you need to answer just two questions:

1. What's the lowest value for the input and what's the corresponding output?
2. If I know the output for $n-1$, how can I use it to compute the output for n ?

The recursive definition for function f is then:

- if $n = \text{lowest}$: $f(n) = \text{output}$
- if $n > \text{lowest}$: $f(n) = \text{some expression based on } f(n-1) \text{ and } n$.

Not all problems on integers can be defined recursively. For example, as far as I know there's no recursive definition for being prime, because whether $n-1$ is prime doesn't in general provide a clue about whether n is prime. So for this example, it's not possible, at least without advanced mathematical knowledge, to answer question 2 above.

Exercise 12.2.1

Write a recursive definition of whether a positive integer is even, i.e. divisible by two.

- if $n = \dots$: $\text{even}(n) = \dots$
- if $n > \dots$: $\text{even}(n) = \dots$

Hint Answer

Exercise 12.2.2

Implement it and add more tests. Add print statements if you wish.

```
[1]: from algoesup import test

def even(n: int) -> bool:
    """Write the docstring here."""
    pass

even_tests = [
    # case,           n,      even?
    ('smallest even', 2,     True),
    ('larger even',   100,   True),
    # your tests here
]

test(even, even_tests)
```

Answer

12.3 Length of a sequence

The same approach can be used with a collection of items, if we decrease it by one item at a time until it has the lowest number of items imposed by the preconditions (often, zero).

Let's consider the problem of determining the length of a sequence.

12.3.1 Recursive definition

To define the problem recursively, I must answer two questions:

1. What are the smallest possible input and its output? It's the empty sequence, with length 0.
2. If I know the output for a sequence with one item less, how can I use it to compute the output for the whole sequence? The length of the whole sequence is the length of the shorter sequence plus one, because we only shortened it by one item.

The recursive definition will be something like:

- if *sequence* is empty: $\text{length}(\text{sequence}) = 0$
- if *sequence* isn't empty: $\text{length}(\text{sequence}) = \text{length}(\text{shorter sequence}) + 1$.

For integers, the decrease step did the subtraction $n-1$, but how can we decrease collections by one item? For a *stack*, we pop the top item. For a *queue*, we dequeue the front item. For a *linked list*, we remove the first node by making the head point to the next node. For an array we remove any item and shift the others.

To define recursive functions on general sequences, without depending on the operations of each sequence data type, it's useful to define a sequence recursively:

A sequence S is either empty or it consists of an item (called the **head** of S) followed by another sequence (called the **tail** of S).

The second question can now be rephrased as

2. If I know the output for the tail of sequence S, how can I compute the output for S?

Defining the data recursively allows recursive function definitions to simply follow the shape of the data. For the problem at hand, the length of a sequence is zero if it's empty, otherwise it's one more than the length of its tail.

- if *sequence* is empty: $\text{length}(\text{sequence}) = 0$
- if *sequence* isn't empty: $\text{length}(\text{sequence}) = \text{length}(\text{tail}(\text{sequence})) + 1$.

Exercise 12.3.1

Write a recursive definition for $\text{sum}(\text{numbers})$, where *numbers* is a sequence of integers.

- if *numbers* is empty:
- otherwise:

Hint Answer

12.3.2 Code

The algorithm is the same as the recursive definition, just in slightly different notation, so I proceed directly to the code. I will use lists as the sequence data type.

First I write the three functions that support the recursive definition of lists.

```
[1]: # this code is also in m269_rec_list.py

def head(items: list) -> object:
    """Return the first item of the list.

    Preconditions: items isn't empty
    """
    return items[0]

def tail(items: list) -> list:
    """Return the list without the first item.

    Preconditions: items isn't empty
    """
    return items[1:]

def is_empty(items: list) -> bool:
    """Return True if and only if the list is empty."""
    return items == []
```

Now I can implement the recursive definition of length.

```
[2]: def length(items: list) -> int:
    """Return the number of objects in the list."""
    if is_empty(items):
        return 0
    else:
        return length(tail(items)) + 1
```

The last line of code does the decrease, recur and combine steps in one expression. Decreasing the input is done by computing its tail. The combination step increments the solution of the decreased problem by one; the head of the sequence isn't needed for this problem, but that's an exception.

Let's quickly check the implementation.

```
[3]: length([])
[3]: 0
```

```
[4]: length(["one", "two", "three"])
[4]: 3
```

12.3.3 Mistakes

For a recursive definition and the corresponding algorithm to be correct, there must be a base case at which the recursion stops and the recurrence relation must progress towards the base case by decreasing the input.

Without the base case the algorithm doesn't stop. The factorial function will happily decrease n by one forever if there's no check for $n = 0$ (and if the call stack were unlimited).

Consider this implementation of the length function, which lacks the base case.

```
[5]: def wrong_length(items: list) -> int: # noqa: D103
    return wrong_length(tail(items)) + 1
```

What will happen if this Python function is called for a list with three items?

It may look like this function will raise an error after three calls, because the tail operation is undefined for the empty list, according to its preconditions.

However, it's implemented with slicing from index 1. In Python, slicing returns an empty sequence if the indices are out of bounds. So, slicing the empty list (which doesn't have index 1) produces the empty list.

```
[6]: tail([])
[6]: []
```

Function `wrong_length` will keep attempting to decrease the empty list and only stop when the call stack's memory is exhausted.

```
[7]: wrong_length([1, 2, 3])
-----
→-----  
RecursionError
    ↗call last)
Cell In[7], line 1
----> 1 wrong_length([1, 2, 3])

Cell In[5], line 2, in wrong_length(items)
    1 def wrong_length(items: list) -> int: # noqa: D103
    ----> 2     return wrong_length(tail(items)) + 1

Cell In[5], line 2, in wrong_length(items)
    1 def wrong_length(items: list) -> int: # noqa: D103
```

(continues on next page)

(continued from previous page)

```

----> 2      return wrong_length(tail(items)) + 1

[... skipping similar frames: wrong_length at line 2 (2974 times)

Cell In[5], line 2, in wrong_length(items)
    1 def wrong_length(items: list) -> int: # noqa: D103
----> 2      return wrong_length(tail(items)) + 1

RecursionError: maximum recursion depth exceeded

```

12.4 Inspecting sequences

This section shows how to implement other operations that inspect sequences, in order to illustrate some finer points of recursion.

12.4.1 Maximum

Let's imagine that we have a *maximum function* $\text{max}(x, y)$ that only takes two numbers. We can use it to find the maximum of a sequence. As usual, we start with the questions leading to a recursive definition.

1. What's the smallest input and its output?
2. What's the output for sequence S, given the output for the tail of S?

Applied to this problem, they are:

1. What's the smallest number sequence and its largest number?
2. What's the largest number in S, knowing the largest number in the tail of S?

1. The smallest input is a sequence of length one; the output is the head.
2. The largest number in S is the larger of the head and the largest of the tail.

This is an example where the base case isn't the empty sequence, because the largest number of an empty sequence is undefined. It's also an example of a more complicated combination step that involves comparing the head with the solution for the tail to determine the solution for the whole sequence.

Given that we already defined the length operation, we can use it. The recursive definition can be written like this:

- if $\text{length}(numbers) = 1$: $\text{maximum}(numbers) = \text{head}(numbers)$
- otherwise: $\text{maximum}(numbers) = \text{max}(\text{head}(numbers), \text{maximum}(\text{tail}(numbers)))$.

Most recursive algorithms have similar base cases, decrease and recur steps; it's the combination step that mostly determines what is being computed. The recursive definitions of sum and length in the previous section only differ in the combination step.

Exercise 12.4.1

Complete the function header, add a docstring and write the function body, following the recursive definition above. Use Python's `len` and `max` functions and the `head` and `tail` functions in file `m269_rec_list.py`.

```
[1]: from algoesup import test

%run -i ../m269_rec_list

def maximum(numbers):
    pass

maximum_tests = [
    # case,           numbers,           maximum
    ('shortest input', [5],             5),
    ('all equal',      [-1, -1],        -1),
    ('ascending',      [-1, 0, 3],       3),
    ('descending',     [4, 2, 1],        4),
    ('unsorted',       [2, 4, 3, 4, 1],  4)
]

test(maximum, maximum_tests)
```

Answer

12.4.2 Membership

Let's now turn our attention to the membership problem: `has(items, item)` is true if and only if `item` is a member of sequence `items`.

I start with the usual two questions.

1. What's the smallest input and the corresponding output? It's the empty sequence, which doesn't have any item, so the output is false.
2. What's the output for sequence S, given the output for the tail of S? The output is true if the item is in the tail or the item is the head.

The recursive definition can be written like this:

- if `items` is empty: `has(items, item) = false`
- otherwise: `has(items, item) = has(tail(items), item) or head(items) is item.`

This definition follows the usual base case, decrease, recur and combine steps. Here, the combination involves a disjunction.

However, it's pointless to search the tail for the item if it's the head. In languages with *short-circuit disjunction* we just have to swap the order of the disjunction operands. Another way is to make the head item a base case.

- if *items* is empty: $\text{has}(\text{items}, \text{item}) = \text{false}$
- otherwise if *item* is $\text{head}(\text{items})$: $\text{has}(\text{items}, \text{item}) = \text{true}$
- otherwise: $\text{has}(\text{items}, \text{item}) = \text{has}(\text{tail}(\text{items}), \text{item})$.

This example illustrates that some recursive algorithms have more than one base case, not just the smallest input.



Note: A base case is any input that doesn't require a recursive call.

As usual, the recursive definition can be directly translated to Python code.

```
[2]: %run -i ../m269_rec_list

def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items."""
    if is_empty(items):
        return False
    elif head(items) == item:
        return True
    else:
        return has(tail(items), item)

has([1, 3, -1], 2)
[2]: False
```

There's no combination step after the recursive step, i.e. after calling `has`. When the recursive call is the last operation in a function, it's called **tail recursion**.

When a function is tail recursive, the result of the recursive call is passed directly to the caller, which in turn passes it directly to its caller, and so on. Because there's no combination step, the result of the final recursive call can be passed directly to the first call. The interpreter can keep only the current frame on the stack: there's no point in storing all local variables of all intermediate calls, as they won't be used – the code does nothing after the recursive call.

Tail recursion can thus be implemented in a very efficient way without a stack that keeps growing. The Python and Java interpreters don't do tail-recursion optimisation; functional language interpreters usually do and for other languages you may be able to turn the optimisation on or off.

Exercise 12.4.2

Write a recursive definition for function `ascending(numbers)`, which returns true if and only if each number in sequence `numbers` is smaller than the next number or if $|numbers| < 2$.

- if ...: `ascending(numbers) = ...`
- otherwise ...

Hint Answer

12.4.3 Indexing

Functions with more than one argument may require several of them being decreased or being checked for a base case.

Consider the *indexing operation*: return the value of a sequence `items` at a given `index`. Recall that the precondition is $0 \leq index < |items|$, which implies that the sequence can't be empty.

The smallest possible inputs are `index = 0` and a sequence with just one item. However, it's possible to access the first item of any non-empty sequence. So the base case is just `index = 0`, not that the sequence has one item. The corresponding output is the head.

- if `index = 0`: `value(items, index) = head(items)`

As for the recurrence relation, it must define the n -th item of a sequence in terms of some item of the tail: it's the $n - 1$ -th item, e.g. the fifth element of a sequence is the fourth element of its tail.

- otherwise: `value(items, index) = value(tail(items), index - 1)`

This problem illustrates that a recursive definition may only check one of the arguments for the base case but may have to decrease two or more arguments. It's also another example of tail recursion.

Exercise 12.4.3

Write the code and some tests.

```
[3]: from algoesup import test

%run -i ../m269_rec_list

def value(items, index) -> ...:
    # add type annotations and a docstring
    pass

value_tests = [
    # your tests here
]

test(value, value_tests)
```

Answer

12.5 Creating sequences

This section shows further examples of recursion applied to sequences. The operations here return a new sequence, instead of a single value.

12.5.1 Prepend

Sequences are decomposed into a head and a tail; therefore they can be composed using prepend (insert at index 0). The recursive definition of a sequence can be restated as:

A sequence S is either empty or prepend(head(S), tail(S)).

I put the sequence as the second argument of the prepend operation to reinforce that the prepended item comes first in the output sequence. The operation can be implemented in two ways, depending on whether we want to modify the sequence ...

```
def prepend(item: object, items: list) -> None:  
    items.insert(0, item)
```

... or create a new one. I'll take that approach.

```
[1]: # this code is also in m269_rec_list.py  
  
def prepend(item: object, items: list) -> list:  
    """Return a new list with item as head and items as tail."""  
    return [item] + items
```

12.5.2 Linear search

Section 12.4.1 showed a recursive way of finding the maximum number in a sequence. The same approach can be used to implement a linear search for all candidates that satisfy some conditions. Let's take the questions in order:

1. What's the smallest input and the corresponding output?

The smallest input is the empty sequence and so is the output.

2. If we know the output for the tail of S, what's the output for S?

If we found all matching items in the tail, then we must prepend the head if it also satisfies the conditions. Otherwise, the solutions for S are just the solutions for the tail of S. By using prepend, the solutions are in the same order as the candidates.

The recursive definition is:

- if *candidates* is empty: $\text{search}(\text{candidates}) = ()$
- otherwise if $\text{head}(\text{candidates})$ doesn't satisfy the conditions: $\text{search}(\text{candidates}) = \text{search}(\text{tail}(\text{candidates}))$

- otherwise: $\text{search}(\text{candidates}) = \text{prepend}(\text{head}(\text{candidates}), \text{search}(\text{tail}(\text{candidates})))$.

The algorithmic pattern is a direct translation of the definition:

1. if candidates is empty:
 1. let solutions be ()
2. otherwise:
 1. if $\text{head}(\text{candidates})$ doesn't satisfy the conditions:
 1. let solutions be $\text{search}(\text{tail}(\text{candidates}))$
 2. otherwise:
 1. let solutions be $\text{prepend}(\text{head}(\text{candidates}), \text{search}(\text{tail}(\text{candidates})))$.

Step 2.2.1 decreases the input, recurs and combines the head with the solution for the tail, while step 2.1.1 doesn't do any combination. A slightly shorter version of step 2 is:

2. otherwise:
 1. let solutions be $\text{search}(\text{tail}(\text{candidates}))$
 2. if $\text{head}(\text{candidates})$ satisfies the conditions:
 1. let solutions be $\text{prepend}(\text{head}(\text{candidates}), \text{solutions})$

Exercise 12.5.1

Use the above pattern (either version of step 2) to implement a Python function that returns all positive numbers in a sequence in the order they occur. Add tests.

```
[2]: from algoesup import test

%run -i ../m269_rec_list

def positive(numbers: list) -> list:
    """Return a new list of all positive numbers in the input.

    Preconditions: all elements of numbers are integers or floats
    Postconditions:
        the output's elements are in the same order as in the input
    """
    pass

positive_tests = [
    # case,           numbers,           positive
    ('shortest input', [],                []),
    # new tests:
]
```

(continues on next page)

(continued from previous page)

```
test(positive, positive_tests)
```

Hint Answer

12.5.3 Append and insert

Other basic operations that construct sequences can be defined in terms of prepend, e.g. the $\text{append}(\text{items}, \text{item})$ operation, which produces a new sequence with item at the end.

Obtaining a recursive definition is the same old story. I first think about the smallest input and its output. For this operation, the base case is the empty sequence. The output is a sequence with the given item only. Appending an item to an empty sequence is the same as prepending it.

As for the non-empty case, if I've appended the item to the tail, I only need to put the head back on at the start. The definition is:

- if items is empty: $\text{append}(\text{items}, \text{item}) = \text{prepend}(\text{item}, \text{items})$
- otherwise: $\text{append}(\text{items}, \text{item}) = \text{prepend}(\text{head}(\text{items}), \text{append}(\text{tail}(\text{items}), \text{item}))$.

Exercise 12.5.2

Define operation $\text{reverse}(\text{items})$ recursively, using the append function. For example, $\text{reverse}(1, 2, 3) = (3, 2, 1)$ and $\text{reverse}() = ()$.

Hint Answer

Exercise 12.5.3

Define $\text{insert}(\text{items}, \text{item}, \text{index})$ recursively, using prepend. For example, $\text{insert}(1, 2), 1.5, 1) = (1, 1.5, 2)$. Assume $0 \leq \text{index} < |\text{items}|$.

Hint Answer

12.6 Avoiding slicing

Recursive algorithms that use the tail operation on arrays aren't very efficient because it takes linear time to slice an array.

To avoid slicing, we add two parameters to the recursive function: the start and end indices of the slice to be processed. Each recursive call reduces the range of the indices instead of creating a new smaller sequence. In other words, we keep track of the slice but don't extract it from the sequence. Let's see an example.

12.6.1 Problem definition

Consider the *membership function*.

Function: membership

Inputs: $items$, a sequence; $item$, an object

Preconditions: true

Output: $exists$, a Boolean

Postconditions: $exists$ is true if and only if there's an integer $index$ such that $items[index] = item$

A more general membership operation allows the user to specify in which slice of the input sequence the item should be searched.

Function: slice membership

Inputs: $items$, a sequence; $item$, an object; $start$, an integer; end , an integer

Preconditions: $0 \leq start \leq end \leq |items|$

Output: $exists$, a Boolean

Postconditions: $exists$ is true if and only if there's an integer $index$ such that $items[index] = item$ and $start \leq index < end$

This new operation can be seen as a combination of the *slicing* and membership operations. As such, it has the same preconditions as the slicing operation and follows the same convention of not including the end item in the slice.

The less general membership function can now be defined more simply: the item exists in the sequence if it exists in the slice comprising the whole sequence.

Function: membership

Inputs: $items$, a sequence; $item$, an object

Preconditions: true

Output: $exists$, a Boolean

Postconditions: $exists = \text{slice membership}(items, item, 0, |items|)$

12.6.2 Recursive definition

For the recursive definitions, I use the shorter function name ‘has’ instead of ‘membership’.

Function $\text{has}(items, item)$ was defined as follows in [Section 12.4.2](#):

- if $items$ is empty: $\text{has}(items, item) = \text{false}$
- otherwise if $item$ is $\text{head}(items)$: $\text{has}(items, item) = \text{true}$
- otherwise: $\text{has}(items, item) = \text{has}(\text{tail}(items), item)$.

The new function $\text{slice_has}(items, item, start, end)$ is defined in a similar way, but works on the slice given by the start and end indices instead of the whole sequence. The first base case is for the slice to be empty.

- if $start = end$: $\text{slice_has}(items, item, start, end) = \text{false}$

The second base case is for the head of the slice to be the sought item.

- otherwise if $items[start] = item$: $\text{slice_has}(items, item, start, end) = \text{true}$

The recurrence relation looks for the item in the tail of the slice, i.e. after the current $start$ index.

- otherwise: $\text{slice_has}(items, item, start, end) = \text{slice_has}(items, item, start + 1, end)$

The recurrence relation is progressing towards the first base case, as it should: increasing the start index gets it closer to being equal to the end index.

Exercise 12.6.1

Here again is the algorithmic pattern for a *recursive linear search* that outputs all items from the input sequence $candidates$ which satisfy some conditions.

1. if $candidates$ is empty:
 1. let $solutions$ be ()
2. otherwise:
 1. let $solutions$ be $\text{search}(\text{tail}(candidates))$
 2. if $\text{head}(candidates)$ satisfies the conditions:
 1. let $solutions$ be $\text{prepend}(\text{head}(candidates), solutions)$

Modify the pattern so that it implements the function $\text{search}(candidates, start, end)$, which only searches the slice $candidates[start:end]$. The precondition is $0 \leq start \leq end \leq |numbers|$.

Hint Answer

Exercise 12.6.2

Here again is the recursive definition of the maximum operation on a non-empty sequence of numbers.

- if $\text{length}(numbers) = 1$: $\text{maximum}(numbers) = \text{head}(numbers)$
- otherwise: $\text{maximum}(numbers) = \max(\text{head}(numbers), \text{maximum}(\text{tail}(numbers)))$

Write a recursive definition for a function $\text{maximum}(numbers, start, end)$ that returns the largest number in the slice $numbers[start:end]$. The precondition is $0 \leq start < end \leq |numbers|$.

Hint Answer

12.6.3 Code

The algorithm and code directly follow the recursive definition, so I skip the algorithm.

```
[1]: from algoesup import test

def slice_has(items: list, item: object, start: int, end: int) -> bool:
```

(continues on next page)

(continued from previous page)

```

"""Return True if and only if item is a member of slice
→items[start:end].

Preconditions: 0 <= start <= end <= len(items)
"""

if start == end:
    return False
elif items[start] == item:
    return True
else:
    return slice_has(items, item, start + 1, end)

def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items."""
    return slice_has(items, item, 0, len(items))

has_tests = [
    # case,           items,     item,   has?
    ('empty list',  [],        3,      False),
    ('last member', [1, 2, 3],  3,      True),
    ('not member',  [1, 2, 3],  4,      False),
]

test(has, has_tests)

```

Testing has...

Tests finished: 3 passed (100%), 0 failed.

If for some reason we don't want to provide the `slice_has` function to the user, we can hide it by nesting it inside the main `has` function. I introduced inner functions in [Section 11.3.3](#) to test a candidate for complicated conditions. An inner function can access the arguments of the outer function. Here, the sequence being searched and the sought item are never modified, so we don't have to constantly pass them to each recursive call. The code becomes slightly shorter.

```
[2]: def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items."""

    def in_slice(start: int, end: int) -> bool:
        """Return True if and only if item is in slice items[start:
→end].

        Preconditions: 0 <= start <= end <= len(items)
        """

        if start == end:
            return False
```

(continues on next page)

(continued from previous page)

```

        elif items[start] == item:
            return True
        else:
            return in_slice(start + 1, end)

    return in_slice(0, len(items))

test(has, has_tests)  # run against the same tests
Testing has...
Tests finished: 3 passed (100%), 0 failed.

```

Exercise 12.6.3

Implement the maximum function, as defined in the previous exercise, using an inner function.

```
[3]: from algoesup import test

def maximum(numbers: list) -> int:
    """Return the largest number in numbers.

    Preconditions: numbers is a non-empty list of integers
    """
    pass

maximum_tests = [
    # case,           numbers,           maximum
    ('shortest input', [5],             5),
    ('all equal',      [-1, -1],        -1),
    ('ascending',      [-1, 0, 3],       3),
    ('descending',     [4, 2, 1],        4),
    ('unsorted',       [2, 4, 3, 4, 1],  4)
]
test(maximum, maximum_tests)
```

Answer

12.7 Multiple recursion

Recursive algorithms decrease the input to bring it ‘closer’ to the base case. So far we’ve reduced the size of the input collection by separating one item from it, and making a recursive call on the rest of the collection. Nothing prevents us from dividing the input in a different way, e.g. we

could divide a sequence in two halves, recur into each half and then combine the subsolutions obtained.

12.7.1 Dividing the input

Let's revisit the maximum of a non-empty sequence of numbers. You implemented the following algorithm in a [previous exercise](#): separate the sequence into its head and tail, compute the maximum of the tail and return the larger of it and the head.

1. if $|numbers| = 1$:
 1. let *solution* be $\text{head}(numbers)$
2. otherwise:
 1. let *solution* be $\max(\text{head}(numbers), \text{maximum}(\text{tail}(numbers)))$

An alternative approach divides the sequence in two halves, computes their maxima and returns the larger of the two.

1. let n be $|numbers|$
2. if $n = 1$:
 1. let *solution* be $\text{head}(numbers)$
3. otherwise:
 1. let *middle* be $\text{floor}(n / 2)$
 2. let *largest left* be $\text{maximum}(numbers[0:middle])$
 3. let *largest right* be $\text{maximum}(numbers[middle:n])$
 4. let *solution* be $\max(\text{largest left}, \text{largest right})$

Steps 3.2 and 3.3 make a recursive call each, on the left and right halves of the sequence. This algorithm involves **multiple recursion**; all previous algorithms were examples of **single recursion**. Multiple-recursion algorithms aren't tail recursive, because the first recursive call isn't the last operation of the algorithm.

Splitting a sequence in two equal-sized (or nearly equal-sized) parts is just a matter of convenience because it's easy to calculate the middle index. It's not essential for recursion to work. I could have divided the sequence into a left part with one third of the items and a right part with the remaining two thirds of them, or any other ratio. I could also have divided the sequence in three parts and return the largest of the three maxima. These alternative ways of dividing the input complicate the algorithm and, if we're careless, make it incorrect. Consider the one versus two thirds split.

3. otherwise:
 1. let *third* be $\text{floor}(n / 3)$
 2. let *largest left* be $\text{maximum}(numbers[0:third])$
 3. let *largest right* be $\text{maximum}(numbers[third:n])$
 4. let *solution* be $\max(\text{largest left}, \text{largest right})$

If $n = 2$, then $third = 0$, the left part is empty and the right part is $numbers$. The right part doesn't reduce in size so the recursion doesn't stop. This issue can be solved by handling $n = 2$ as an additional base case.

12.7.2 Designing multiple recursion

When designing algorithms with multiple recursion, the questions to ask yourself are:

1. How can the input be divided in two or more parts? (decrease step)
2. If I know the output for each part, what's the output for the whole input? (combine step)
3. What are the smallest parts that cannot be further divided and their corresponding outputs? (base cases)

Previously there were only two questions: first about the base cases, then about the combine step. The input was always decreased in the same way: remove one item or decrement the input integer by one.

With multiple recursion there are usually several ways to decrease the input, some more appropriate to the problem at hand than others, and each may have different base cases, so it's best to think of them after deciding how to partition the input.

Slicing a sequence in two or more parts takes linear time, whether the sequence is represented with an array or a linked list. We can reuse the technique of keeping track of the slice's start and end instead of extracting the slice. The algorithm for maximum($numbers, start, end$) is as follows.

1. if $start + 1 = end$:
 1. let $solution$ be $numbers[start]$
2. otherwise:
 1. let $middle$ be $start + \text{floor}((end - start) / 2)$
 2. let $largest\ left$ be $\text{maximum}(numbers, start, middle)$
 3. let $largest\ right$ be $\text{maximum}(numbers, middle, end)$
 4. let $solution$ be $\max(largest\ left, largest\ right)$

Step 2.1 obtains the middle index by adding half of the length of the slice to the start index. When $start = 0$ and $end = |numbers|$, the middle index is as in the earlier algorithm:
 $start + \text{floor}((end - start) / 2) = \text{floor}(|numbers| / 2)$.

Exercise 12.7.1

Write a multiple-recursive algorithm for the *membership function* has($items, item, start, end$).

Hint Answer

Exercise 12.7.2

Implement your algorithm of the previous exercise.

```
[1]: from algoesup import test

def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items."""

    def in_slice(start: int, end: int) -> bool:
        """Return True if and only if item is in slice items[start:end].

        Preconditions: 0 <= start <= end <= len(items)
        """
        pass

    return in_slice(0, len(items))

has_tests = [
    # case,           items,     item,   has?
    ('empty list',  [],        3,      False),
    ('last member', [1, 2, 3],  3,      True),
    ('not member',  [1, 2, 3],  4,      False),
]
test(has, has_tests)      # run against the same tests
```

Answer

12.8 Summary

A function (in the mathematical sense) can be defined recursively with one or more **base cases**, for which the output can be directly computed, and one or more **recurrence relations**, which define how the output is obtained by applying the same function (hence the adjective ‘recursive’) to one or more parts of the input.

A **recursive algorithm** does the following:

- check if the input is a base case and, if so, compute the output and stop
- divide the input into smaller parts
- recur over one or more parts to solve the problem for those smaller instances
- combine the partial solutions with the parts not recurred over to obtain the solution for the problem.

A **single-recursion** algorithm only recurs over one part; a **multiple-recursion** algorithm recurs over two or more parts.

If the input is a collection of items, a single-recursion algorithm usually partitions it into an item and the rest of the collection, and recurs over the latter. A multiple-recursion algorithm usually partitions the input collection in two halves, and recurs over both.

A single-recursion algorithm is **tail recursive** if the recursive call is the last operation of the algorithm.

A sequence can be defined recursively as either being empty (the base case), or by prepending an item (the **head**) to a shorter sequence (the **tail**). This allows us to define algorithms on sequences recursively. File `m269_rec_list.py` provides functions `is_empty`, `head`, `tail` and `prepend` for Python lists.

Obtaining the tail of an array or partitioning a sequence in two or more parts requires linear-time slicing. We can avoid it by passing to each recursive call the start and end indices of the slice to be processed.

A programming language interpreter executes functions with the help of a **call stack** where the information about each suspended function call is stored in a so called **stack frame**, so that the suspended function can be resumed after the current call ends. A recursive call is a call to the same function that is currently executing.

There is limited space for the call stack, so making many recursive calls will lead to an error. This happens especially for recursive algorithms that decrease large inputs only by one item at a time. Some interpreters (but not for Python) can execute tail-recursive functions without increasing the call stack.

CHAPTER 13

DIVIDE AND CONQUER

Divide and conquer is a problem-solving strategy that divides the input into two or more parts, solves (conquers) the smaller problem instances then combines the solutions for the parts to obtain the solution for the original input. Divide-and-conquer algorithms are usually written with multiple recursion. The algorithm that finds the maximum of a sequence by *finding the maximum of each half* is an example of divide and conquer.

A **decrease-and-conquer** algorithm is a divide-and-conquer algorithm that creates only one smaller part or that creates multiple parts but only solves one of them. You've seen examples of decrease and conquer in the previous chapter. They all created a single smaller part, e.g. the tail of the input sequence. This chapter will show an example of partitioning the input into two parts but only conquering one of them. Decrease-and-conquer algorithms usually use single recursion. Tail-recursive algorithms can be easily rewritten using iteration.

The previous chapter presented decrease-and-conquer algorithms that reduce the input size just by one. This chapter presents algorithms that decrease the size by a larger amount and are therefore more efficient. You will see that the complexity of a recursive algorithm can be defined recursively.

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn various forms of decrease-and-conquer algorithms, including binary search, and their logarithmic and log-linear complexities.
- Analyse the complexity of algorithms to support software design choices – you will learn how to determine the complexity of some recursive algorithms.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

13.1 Decrease by one

This and the next two sections introduce the three kinds of decrease-and-conquer algorithms, according to how they decrease the input.

Some algorithms decrease the input's size or value by a **constant amount**, usually by one to make sure that the base case isn't skipped over, as I explained [previously](#). All single recursion examples in the previous chapter are examples of decrease-by-one algorithms.

This section explains how to analyse the complexity of such algorithms.

13.1.1 Factorial

Here again is the recursive algorithm for the *factorial*:

1. if $n = 0$:
 1. let *product* be 1
2. otherwise:
 1. let *product* be $\text{factorial}(n-1) \times n$

The complexity of an iterative algorithm is the number of iterations multiplied by the complexity of each one. Similarly, the complexity of a recursive algorithm is the number of calls multiplied by the complexity of each one.

Computing $n!$ involves $n+1$ calls, from $\text{factorial}(n)$ until $\text{factorial}(0)$. Each call only does constant-time operations: one comparison, one assignment and one multiplication (unless n is zero). The complexity is therefore $(n + 1) \times \Theta(1) = \Theta(n)$.

Another way to obtain the complexity is to define it recursively for each case. Let $T(n)$ be the algorithm's complexity for input n . The function name T is an abbreviation for 'time' but any other name would do.

If the input is zero, the algorithm takes constant time (steps 1 and 1.1), so $T(0) = \Theta(1)$. If the input is positive (step 2.1), the algorithm takes whatever time it needs to compute $(n-1)!$ plus some constant time to multiply the result by n , i.e. $T(n) = T(n-1) + \Theta(1)$. The definition of T can be written like for any other recursive function:

- if $n = 0$: $T(n) = \Theta(1)$
- if $n > 0$: $T(n) = T(n-1) + \Theta(1)$.

If we repeatedly expand the right-hand side for $n > 0$ until $n = 0$ we unsurprisingly get the same result.

$$T(n) = T(n-1) + \Theta(1) = T(n-2) + \Theta(1) + \Theta(1) = \dots = T(n-n) + n \times \Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$$

13.1.2 Sequence length

As a further example, let's consider the length of a sequence. The following algorithm is based on the earlier *recursive definition*.

1. if *sequence* is empty:
 1. let *size* be 0
2. otherwise:
 1. let *size* be $\text{length}(\text{tail}(\text{sequence})) + 1$

To make the complexity easier to write, I abbreviate $|\text{sequence}|$ as n . $T(n)$ will be the complexity for an input of size n . The recursive definition is:

- if $n = 0$: $T(n) = \Theta(1)$
- if $n > 0$: $T(n) = \Theta(t) + T(n - 1) + \Theta(1)$.

In the recurrence relation, $\Theta(t)$ is the complexity of obtaining the tail, $T(n - 1)$ is for computing the length of the tail and $\Theta(1)$ is for adding one to result of the recursive call.

For *linked lists*, the tail is obtained in constant time by following two pointers: first to the head, then to the next node. The recurrence relation becomes

$$T(n) = \Theta(1) + T(n - 1) + \Theta(1) = T(n - 1) + \Theta(1).$$

As we've seen for the factorial example, a recursive definition of this form entails $T(n) = \Theta(n)$. Calculating the length of a linked list takes linear time.

However, on arrays the tail operation takes linear time due to slicing. The recurrence relation becomes

$$T(n) = \Theta(n) + T(n - 1) + \Theta(1) = T(n - 1) + \Theta(n).$$

Successively expanding T we get

$$T(n) = T(n-1) + \Theta(n) = T(n-2) + \Theta(n-1) + \Theta(n) = \dots = T(0) + \Theta(1) + \Theta(2) + \dots + \Theta(n).$$

Since $T(0) = \Theta(1)$, we get

$$T(n) = \Theta(1) + \Theta(1) + \Theta(2) + \dots + \Theta(n) = \Theta(1) + \Theta(2) + \dots + \Theta(n).$$

The sum of the first n positive integers is $(n^2 + n) / 2$. We therefore have

$$T(n) = \Theta(1) + \dots + \Theta(n) = \Theta(1 + \dots + n) = \Theta((n^2 + n) / 2) = \Theta(n^2)$$

because the growth rate is given by the fastest-growing term, ignoring constant factors like one half. Computing the length of an array with the tail operation takes quadratic time.



Info: MU123 Unit 9 Section 1.1 and MST124 Unit 10 Section 4.1 explain how the formula for $1 + \dots + n$ is derived.

Both the factorial and the length have base case $n = 0$, but the recursive definition of the complexity is the same for whatever smallest input, as long as it takes constant time to compute its solution.



Note: If the complexity is defined by $T(s) = \Theta(1)$, where s is the smallest size or value, and $T(n) = T(n - 1) + \Theta(1)$ for $n > s$, then $T(n) = \Theta(n)$. If instead $T(n) = T(n - 1) + \Theta(n)$, then $T(n) = \Theta(n^2)$.

Linked lists are better suited than arrays for recursive algorithms on sequences, because they support the tail and prepend operations in constant time without having to copy or shift items, as you've seen when *implementing stacks*.

13.2 Decrease by half

Progressing towards the base case one item at a time is slow. This and the next section show more efficient decrease-and-conquer algorithms that decrease the input each time by a substantial amount.

One way is to decrease the input by a **constant factor** f rather than by a constant amount a , i.e. the size or value is reduced to n / f rather than $n - a$. Usually $f = 2$, i.e. the input is decreased by half.

13.2.1 Problem

Consider the *exponentiation operation*: compute b^n for integers b (the base) and n (the exponent), with non-negative n .

A decrease-by-one definition is similar to the factorial:

- if $n = 0$: $b^n = 1$
- if $n > 0$: $b^n = b^{n-1} \times b$.

This effectively multiplies b by itself n times, in linear time. The algorithm follows directly from the recursive definition, so I skip to the code. For the mathematical notation, it's convenient to have single-letter variable names, but when writing code they should be descriptive.

```
[1]: def power_by_one(base: int, exponent: int) -> int:  
    """Return the base to the power of the exponent.  
  
    Preconditions: exponent >= 0  
    """  
    if exponent == 0:  
        return 1  
    else:  
        return power_by_one(base, exponent - 1) * base
```

(continues on next page)

(continued from previous page)

```
power_by_one(3, 20) == 3**20 # test with Python's power operator
```

[1]: True

13.2.2 Algorithm

Using some properties of exponentiation, we can halve the exponent instead of decreasing it by one, to do fewer multiplications. For example, $4^6 = 4^3 \times 4^3$. Assuming 4^3 requires three multiplications, we need four instead of six multiplications to obtain 4^6 . By halving the exponent we get twice the same expression and only compute it once. If the exponent is odd, we need one extra multiplication, e.g. $4^5 = 4^2 \times 4^2 \times 4$. The general recursive definition is:

- if $n = 0$: $b^n = 1$
- if $n > 0$ and is even: $b^n = b^{n/2} \times b^{n/2}$
- if n is odd: $b^n = b^{(n-1)/2} \times b^{(n-1)/2} \times b$.

The definition has one base case and two recurrence relations. They cover all possible values of n . For example, if $n = 1$ then the last case applies and we have $b^1 = b^0 \times b^0 \times b = 1 \times 1 \times b = b$.

Here's the algorithm, with an auxiliary variable for the subsolution to avoid recomputing it.

1. if $n = 0$:

 1. let *solution* be 1

2. otherwise:
 1. let *subsolution* be *power*(*b*, $\text{floor}(n / 2)$)
 2. if $n \bmod 2 = 0$:
 1. let *solution* be *subsolution* \times *subsolution*
 3. otherwise:
 1. let *solution* be *subsolution* \times *subsolution* \times *b*

The last steps could also be written as:

2. otherwise:
 1. let *subsolution* be *power*(*b*, $\text{floor}(n / 2)$)
 2. let *solution* be *subsolution* \times *subsolution*
 3. if $n \bmod 2 = 1$:
 1. let *solution* be *solution* \times *b*

I prefer the first alternative: its intent is clearer, in my opinion.

13.2.3 Complexity

Each recursive call takes constant time because it does at most four arithmetic operations: integer division, modulo and one or two multiplications. The complexity is therefore $r \times \Theta(1) = \Theta(r)$, where r is the number of recursive calls.

Each extra recursive call can handle up to double the value of the exponent. With r recursive calls, the algorithm can handle any n up to 2^r . You've seen this exponential growth rate *before*: every item added to a set doubles the number of subsets the set has.

What we're really interested in is the inverse relationship: how r grows in terms of the input n . The inverse of the exponential is the logarithm: $\log_b b^y = y$ for any real number $b > 1$. The notation $\log_b n$ is read 'logarithm of n to base b '. For this problem, $n = 2^r$, so $\log_2 n = \log_2 2^r = r$. We say that the exponentiation algorithm has **logarithmic complexity** $\Theta(\log_2 n)$.

Actually, the base of the logarithm doesn't matter for complexity analysis because it has been shown that the logarithms of the same number in different bases only differ by a constant factor. Thus, $\log_a n$ and $\log_b n$ have the same growth rate for any bases a and b , and we just write $\Theta(\log n)$ without any base.



Info: Logarithms are covered in MU123 Unit 13 Sections 4 and 5, and in MST124 Unit 3 Section 4.

The safest way to analyse recursive algorithms is to write the recursive definition of T and see which pattern it follows. For this algorithm we have:

- if $n = 0$: $T(n) = \Theta(1)$
- if $n > 0$: $T(n) = T(\text{floor}(n / 2)) + \Theta(1)$.

Whether the algorithm halves an even exponent or halves and rounds down an odd exponent makes no difference to the complexity, so we can simply write $T(n) = T(n / 2) + \Theta(1)$. It has been proven that such a recursive definition leads to $T(n) = \Theta(\log n)$.



Note: If $T(0) = \Theta(1)$ and $T(n) = T(n / 2) + \Theta(1)$, then $T(n) = \Theta(\log n)$.

When introducing *run-time measurements*, I noted that although we assumed b^n to take $\Theta(n)$ to do n constant-time multiplications, Python's interpreter took less than linear time to compute it. We henceforth assume exponentiation takes logarithmic time in n .

13.2.4 Code and performance

Let's implement the decrease-by-half approach.

```
[2]: def power_by_half(base: int, exponent: int) -> int:
    """Return the base to the power of the exponent.

    Preconditions: exponent >= 0
    """
    if exponent == 0:
        return 1
    else:
        subsolution = power_by_half(base, exponent // 2)
        if exponent % 2 == 0:
            return subsolution * subsolution
        else:
            return subsolution * subsolution * base

power_by_half(3, 20) == 3**20
[2]: True
```

Since the complexity depends on the exponent only, to measure the run-time I use always the same base, start with a not too small exponent and double it each time.

```
[3]: exponent = 20
while exponent <= 200:
    %timeit -r 5 -n 10_000 power_by_one(3, exponent)
    exponent = 2 * exponent

1.21 µs ± 246 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
 ↴each)
1.83 µs ± 45 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
 ↴each)
3.86 µs ± 13.4 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
 ↴each)
10.5 µs ± 133 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
 ↴each)
```

The doubling of the run-time confirms the algorithm is linear in the exponent. Now the decrease-by-half approach.

```
[4]: exponent = 20
while exponent <= 200:
    %timeit -r 5 -n 10_000 power_by_half(3, exponent)
    exponent = 2 * exponent

342 ns ± 4.37 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
 ↴each)
```

(continues on next page)

(continued from previous page)

```
413 ns ± 1.22 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops↳each)  
496 ns ± 0.582 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops↳each)  
592 ns ± 0.717 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops↳each)
```

The run-time increases by a fixed amount each time because doubling the exponent requires a single extra multiplication.



Note: If doubling the input size increases the run-time by a fixed amount, then the complexity is logarithmic.

An exponential function with integer base greater than one grows very fast; the logarithm function with the same base thus grows very slowly. For example, 2^{20} is about one million, so computing $b^{1,000,000}$ takes just $\log_2 1,000,000 \approx 20$ recursive calls! Even if each one does two multiplications (the worst case), 40 multiplications is far better than doing a million of them. The efficiency gain is tremendous, even compared to a linear algorithm. If you find a logarithmic algorithm for a problem, you're on to a winner.

13.3 Variable decrease

The third kind of decrease-and-conquer algorithm reduces the size or value of the input by a **variable amount**, i.e. a possibly different amount in every recursive call or iteration. An example is Euclid's algorithm for the greatest common divisor (GCD), which you saw in the BBC programme in the *first week*.

13.3.1 Problem

The GCD of two positive integers is the greatest positive integer that divides both without a remainder, e.g. $\text{gcd}(5, 15) = 5$ and $\text{gcd}(3, 7) = 1$. It can be formulated as a search problem: given positive integers a and b , find the largest integer n that is a factor of a and of b .

Exercise 13.3.1

Before looking at Euclid's algorithm, *outline* a brute-force search algorithm.

Hint Answer

13.3.2 Algorithm

The BBC programme gave a geometric explanation of Euclid's algorithm: the GCD is the side of the largest square tile that covers a rectangle of a given width and length. If you want to watch it again, it's in [part 1](#) from about 5:30 to 8:30.

We start with a rectangle of area $width \times length$, with $width < length$. We then fill the rectangle as much as possible with square tiles of area $width \times width$. The rectangle that remains to be filled has now area $width \times (length \bmod width)$. By definition of the modulo operation, $length \bmod width < width$, so if we take the length to always be the longest side, the width of the original rectangle is now the length of the remaining rectangle.

We repeat the process until no rectangle to be filled remains, i.e. when one of its sides is zero. At that point, the side of the last tile used is the GCD.

If we define the operation so that the first argument is the width and the second the length, and we allow the width to become zero, the definition is:

- if $width = 0$: $\text{gcd}(width, length) = length$
- if $width > 0$: $\text{gcd}(width, length) = \text{gcd}(length \bmod width, width)$.

The programme's example is: $\text{gcd}(150, 345) = \text{gcd}(45, 150) = \text{gcd}(15, 45) = \text{gcd}(0, 15) = 15$.

The recursive algorithm is trivial: it just follows the recursive definition.

1. if $width = 0$:
 1. let *factor* be *length*
2. otherwise:
 1. let *factor* be $\text{gcd}(length \bmod width, width)$

Being a tail-recursive algorithm, the output for the last call is the output for the initial call because there's no combination step. The algorithm just decreases the input until it computes the solution for the base case. This can be done iteratively too.

1. while $width > 0$:
 1. let *new width* be $length \bmod width$
 2. let *length* be *width*
 3. let *width* be *new width*
2. let *factor* be *length*

As Euclid's algorithm shows, decrease and conquer is an old technique that predates computers. All proper books on algorithms include an ancient algorithm.

13.3.3 Complexity

In the best case, the length is a multiple of the width, i.e. the modulo is zero, and the algorithm computes the result in two calls or one iteration, e.g. $\text{gcd}(4, 400) = \text{gcd}(0, 4) = 4$. The best-case complexity is $\Theta(1)$.

In the worst case, it can be proven that the second argument, the length, decreases by at least half in each recursive call or iteration, as the BBC's example illustrates, so this algorithm is logarithmic too.

However, the modulo can decrease the length to substantially less than one half. In the example, the length decreases at one point from 150 to 45, to less than a third, and then again from 45 to 15, exactly one third. The number of calls or iterations is at most the logarithm of the initial length, but for many input values it's much less.

When we don't know the exact growth rate for each input, e.g. because each iteration or recursive call decreases a value by a variable amount, but we know its upper bound, we use the **Big-Oh** notation. For this algorithm, we write $O(\log \text{length})$ instead of $\Theta(\log \text{length})$.

The difference between using Big-Oh and Big-Theta is like saying 'Bob is at most 6 feet tall' and 'Bob is exactly 6 feet tall': the former gives an upper bound whereas the latter is precise.



Note: For decrease-and-conquer algorithms with a variable decrease, use Big-Oh notation instead of Big-Theta.

The Big-Oh notation can be used for the best- and worst-case complexity. For example, if the best-case complexity is $O(n)$, this means that in the best case the algorithm takes at most linear time in n (whatever n means for the problem at hand), but it may take less time, e.g. logarithmic time, we just don't know for which inputs. Normally, the best-case scenario is quite clear and the complexity can be stated precisely, with the Big-Theta notation.

The worst-case complexity is always an upper bound of the best-case complexity, so in many websites and texts you'll read statements of the form 'this algorithm has complexity $O(\dots)$ '. What those authors often mean is that the algorithm has worst-case complexity $\Theta(\dots)$ and some lower complexity for the best case, which they're not interested in.

Every growth rate is an upper bound for a slower growth rate, e.g. 2^n grows faster than n^2 which in turn grows faster than n . So an algorithm with complexity $\Theta(n)$ also has complexity $O(n^2)$ and $O(2^n)$. Although that's technically correct, it's useless information. It's akin to saying that Bob is at most 3 metres tall: it really doesn't give a clue about Bob's real height. If you have to use Big-Oh notation because you can't state the best- or worst-case complexity precisely for all inputs, then give an upper bound as low as you can.

13.4 Binary search

In Chapter 11 you saw that searching for an item x in an ascending sequence is often faster than in an unsorted sequence because we can stop when the current item is larger than x : at that point we know x can't be in the rest of the sequence.

We can search in a sorted sequence much faster with **binary search**. It's a generate-and-test algorithm that generates very few candidates because it decreases the search space by half after testing the current candidate. Binary search is a decrease-and-conquer algorithm and not an exhaustive search because it doesn't go through all candidates.

13.4.1 Recursive, with slicing

Binary search implements the membership operation on sorted sequences. If I'm looking for 4 in the ascending sequence (... , 5, ...), then the number must be in the left part, before the 5, if it exists. If the sequence is in descending order instead, then the 4 must be in the right part, after the 5. More generally, inspecting the middle element of a sorted sequence will always discard half of the items no matter where the sought item is and whether the order is ascending or descending.

Algorithm

The algorithm stops when it reaches one of two base cases: either the middle element is the sought element or the sequence is empty. Here's the recursive definition for $\text{has}(\text{items}, \text{item})$, assuming that the sequence is in ascending order.

- if items is empty: $\text{has}(\text{items}, \text{item}) = \text{false}$
- if $\text{item} = \text{middle element of } \text{items}$: $\text{has}(\text{items}, \text{item}) = \text{true}$
- if $\text{item} < \text{middle element of } \text{items}$: $\text{has}(\text{items}, \text{item}) = \text{has}(\text{left half of } \text{items}, \text{item})$
- if $\text{item} > \text{middle element of } \text{items}$: $\text{has}(\text{items}, \text{item}) = \text{has}(\text{right half of } \text{items}, \text{item})$

I defined the function in informal terms to convey the gist of binary search. The algorithm computes the middle element and slices each half as done before for [computing the maximum](#).

1. let n be $|\text{items}|$
2. if $n = 0$:
 1. let exists be false
3. otherwise:
 1. let middle be $\text{floor}(n / 2)$
 2. let middle item be $\text{items}[\text{middle}]$
 3. if $\text{middle item} = \text{item}$:
 1. let exists be true
 4. otherwise if $\text{item} < \text{middle item}$:
 1. let exists be $\text{has}(\text{items}[0:\text{middle}], \text{item})$
 5. otherwise:
 1. let exists be $\text{has}(\text{items}[\text{middle} + 1:n], \text{item})$

Contrary to the [multiple recursion membership algorithm](#), this one searches only one of the halves: it's a single-recursion algorithm. The difference between multiple and single recursion is not about how many recursive calls are in the algorithm, but how many are executed.

Steps 2 and 3.3 check for the base cases; steps 3.4.1 and 3.5.1 decrease the sequence and recur. There's no combination step, so the algorithm is tail recursive: the last step is either 3.4.1 or 3.5.1.

Neither half includes the middle item (remember that the last index of a slice isn't included), so each recursive call does indeed reduce the size of the input. We don't need another base case.

Exercise 13.4.1

How would you change the algorithm if the sequence were in descending order? Just describe the change briefly: don't write the full algorithm.

Answer

Complexity

In the best-case scenario, the middle element is the sought one: the algorithm takes constant time.

In a worst-case scenario, the sequence doesn't contain the item. We can define the complexity $T(n)$ recursively, using $n = |items|$. The base case (steps 2 and 2.1) takes constant time, so $T(0) = \Theta(1)$. For a non-empty sequence, each call takes

- $\Theta(1)$ for steps 3.1 to 3.3 (which fails in the worst case) and 3.4 or 3.5
- $\Theta(n / 2)$ for slicing half the sequence in either 3.4.1 or 3.5.1
- $T(n / 2)$ for the recursive call in the same step.

Putting it all together:

- if $n = 0$: $T(n) = \Theta(1)$
- if $n > 0$: $T(n) = \Theta(1) + \Theta(n / 2) + T(n / 2) = T(n / 2) + \Theta(n)$.

It has been proven that this leads to $T(n) = \Theta(n)$.



Note: If $T(0) = \Theta(1)$ and $T(n) = T(n / 2) + \Theta(n)$, then $T(n) = \Theta(n)$.

13.4.2 Recursive, without slicing

Binary search with slicing is no better than linear search in terms of complexity and is much worse in terms of memory due to all the intermediate slices created. To make the algorithm more efficient, I keep track of the start and end indices of the slice to [avoid creating it](#).

Here's the recursive binary search algorithm for membership function `has(items, item, start, end)`, with $0 \leq start \leq end \leq |items|$.

1. if $start = end$:
 1. let *exists* be false
2. otherwise:
 1. let *middle* be $start + \text{floor}((end - start) / 2)$
 2. let *middle item* be $items[middle]$

3. if *middle item* = *item*:
 1. let *exists* be true
4. otherwise if *item* < *middle item*:
 1. let *exists* be has(*items*, *item*, *start*, *middle*)
5. otherwise:
 1. let *exists* be has(*items*, *item*, *middle* + 1, *end*)

I code the algorithm with an inner function to hide the extra arguments from the user. I add a print statement that shows the sequence searched in each call. Feel free to uncomment it and to also print the middle item.

```
[1]: def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items.

    Preconditions:
    - items is in ascending order
    - item is comparable to all members of items
    """
    ...

    def in_slice(start: int, end: int) -> bool:
        """Return True if and only if item is in slice items[start:end].

        Preconditions: 0 <= start <= end <= len(items)
        """
        # print('Searching', item, 'in', items[start:end])
        if end == start:
            return False
        else:
            middle = start + (end - start) // 2
            middle_item = items[middle]
            if middle_item == item:
                return True
            elif item < middle_item:
                return in_slice(start, middle)
            else:
                return in_slice(middle + 1, end)

    return in_slice(0, len(items))
```

Here are some tests. You may wish to add more, e.g. with all items the same.

```
[2]: from algoesup import test

has_tests = [
    # case,           items,   item,    has?
    ...]
```

(continues on next page)

(continued from previous page)

```

('empty list',          [],     1,      False),
('is before 1 item',   [2],    1,      False),
('is the 1 item',     [1],    1,      True),
('is after 1 item',   [2],    3,      False),
('is before 2 items',  [2, 4], 1,      False),
('is between 2 items', [2, 4], 3,      False),
('is after 2 items',   [2, 4], 5,      False),
('is 1st of 2 items',  [2, 4], 2,      True),
('is 2nd of 2 items',  [2, 4], 4,      True),
]

test(has, has_tests)

Testing has...
Tests finished: 9 passed (100%), 0 failed.

```

Exercise 13.4.2

Write the recursive definition of T for this algorithm's worst case and determine the complexity by checking if the definition has a form seen before.

Hint Answer

You saw brute-force searches that don't generate symmetric candidates in order to shrink the search space by half or better. For example, for the *factorisation problem* we decreased the upper limit of the search space from n to \sqrt{n} . Binary search prunes the search space by half after *each* test, not just at the start of the algorithm. This makes a tremendous difference, as you've seen for the *exponentiation operation*.

13.4.3 Iterative

The recursive algorithm is tail recursive, so an iterative version is due. It's not strictly necessary because a recursive decrease-by-half function doesn't exceed the call stack limit: processing one billion items requires at most $\log_2 10^9 \approx 30$ calls.

An iterative implementation has the same complexity but is marginally faster, as it avoids the constant-time function call overheads. It simply reduces the slice until it's empty or the item is found. I don't repeat the docstring.

```
[3]: def has_iterative(items: list, item: object) -> bool: # noqa: D103
    start = 0
    end = len(items)
    while start < end: # alternative: while start != end
        middle = start + (end - start) // 2
        middle_item = items[middle]
        if middle_item == item:
            return True
        elif item < middle_item:

```

(continues on next page)

(continued from previous page)

```

        end = middle # search left half [start:middle]
    else:
        start = middle + 1 # search right half [middle+1:end]
    return False

test(has_iterative, has_tests)
Testing has_iterative...
Tests finished: 9 passed (100%), 0 failed.

```

Let's compare the run-times for a worst case: the item isn't in the sequence.

```
[4]: # If running this cell in the VCE leads to a crash, set `items` to
      ↪fewer zeros.
items = [0] * 1_000_000_000 # a billion zeros

%timeit -r 5 has(items, 1)
%timeit -r 5 has_iterative(items, 1)

3.08 µs ± 3.09 ns per loop (mean ± std. dev. of 5 runs, 100,000 ↪
loops each)
2.27 µs ± 45.3 ns per loop (mean ± std. dev. of 5 runs, 100,000 ↪
loops each)
```

Doing 30 iterations instead of 30 recursive calls only saves a few microseconds. Recursion, even without tail optimisation, adds a very small overhead. The efficiency of a search algorithm depends on how it prunes the search space, not on whether it's recursive or iterative.



Note: Recursion isn't inherently inefficient.

13.5 Binary search variants

Binary search can be adapted to other problems, for example if we don't know the value being searched for, only some property of it.

To design a binary search we must answer these questions:

1. When can we know the search is unsuccessful and stop?
2. When can we know the search is successful and stop?
3. How do we decide whether to search the left or the right half of the sequence?

For the basic binary search in the previous section, the answers are:

1. When the sequence is empty.

2. When the middle item is the sought item.
3. If the sought item is smaller than the middle item, search the left half; otherwise search the right half.

13.5.1 Transition

Consider the problem of finding the transition between negative and positive numbers in an ascending sequence. More precisely, we want the index of the first positive number. Let's assume there's always at least one. We don't know its value: it might be 1, 5486, or anything else. Still, we can use binary search to find it.



Info: This problem is inspired by LeetCode problem [278](#).

We're assuming there's a positive number in the sequence, so the search is always successful and question 1 doesn't apply.

What's the answer to question 2, i.e. when can we stop, having found a positive number? Can we stop when the middle number is positive?

No we can't, because there might be other positive numbers to its left. We only stop when the current slice has a single number: it must be positive.

What's the answer to question 3? How do we determine which half to search?

If the middle number is not positive, any positive numbers must come after it because the sequence is ascending, so we search the right half. Otherwise we search the left half. We can't exclude the middle item when searching the left half, because it might be the first positive number.

Here's an algorithm for function `first_positive(numbers, start, end)`, with $0 \leq start < end \leq |numbers|$. Contrary to previous binary searches, the input sequence isn't empty (there's at least one positive number), so the start index is strictly smaller than the end index.

1. if $end - start = 1$:
 1. let *first* be $numbers[start]$
2. otherwise:
 1. let *middle* be $start + \text{floor}((end - start) / 2)$
 2. let *middle item* be $numbers[middle]$
 3. if *middle item* > 0 :
 1. let *first* be `first_positive(numbers, start, middle + 1)`
 4. otherwise:

1. let *first* be *first_positive(numbers, middle + 1, end)*

Since we're working on slices, we're following the convention of not including the end index. Step 2.3.1 must therefore set it to *middle* + 1 in order to include the middle item.

This raises the question of whether the slice is always reducing its length. Length one is already handled as a base case by step 1. Let's assume the length is two, i.e. $end - start = 2$. In that case $middle = start + \text{floor}((end - start) / 2) = start + \text{floor}(2 / 2) = start + 1$

which means that the middle number is the second and last number of the slice. It can't be negative because any positive number would have to come after it, but there are no more numbers in a slice of length 2. So, the middle (actually last) number of the two must be positive. The algorithm will execute step 2.3.1 but $middle + 1 = start + 2 = end$, which means that the recursive call will be made on the same slice. To sum up, when the slice has only two numbers, the second, which must be positive, is chosen as the middle number and the recursive call doesn't decrease the slice.

We must handle this input size as a separate base case and choose either the first number, if both are positive, or else the second number.

2. otherwise if $end - start = 2$:

1. if *numbers[start] > 0*:
 1. let *first* be *numbers[start]*
 2. otherwise:
 1. let *first* be *numbers[start + 1]*
3. otherwise:
 1. ...

Next we must analyse the case $end - start = 3$. Again,

$$middle = start + \text{floor}((end - start) / 2) = start + \text{floor}(3 / 2) = start + 1$$

but now this means that $start < middle + 1 < end$. So, whether the algorithm takes the left half (*start* to *middle* + 1) or the right half (*middle* + 1 to *end*), the new slice is smaller than the input slice from *start* to *end* and there's no risk of infinite recursion.



Note: Check that the recursive calls reduce the input's size or value. Any case for which they don't must be handled as a base case.

Exercise 13.5.1

Implement the inner auxiliary function below recursively and run the tests.

```
[1]: from algoesup import test
```

(continues on next page)

(continued from previous page)

```
def first_positive(numbers: list) -> int:
    """Return the first (lowest index) positive integer in numbers.

    Preconditions:
    - numbers is a list of integers in ascending order
    - numbers has a positive integer
    """
    def in_slice(start: int, end: int) -> int:
        """Return the first positive number within numbers[start:end].

        Preconditions: 0 <= start < end <= len(items)
        """
        pass

    return in_slice(0, len(numbers))

first_positive_tests = [
    # case,           numbers,          first
    ('one number',   [1],             1),
    ('is last',      [-2, -2, 0, 3],  3),
    ('all positive', [2, 3, 4],       2),
    ('all but first', [0, 1, 2, 2, 2, 2], 1),
]
test(first_positive, first_positive_tests)
```

Answer

Exercise 13.5.2

Implement the function iteratively. The docstring isn't repeated.

```
[2]: def first_positive(numbers: list) -> int: # noqa: D103
    pass

test(first_positive, first_positive_tests)
```

Hint Answer

There's a more efficient version with a more general base case: if the start number is positive, no matter how long the slice is, then we've found the first positive integer and we can stop. Once a slice of only positive numbers is obtained, this version stops whereas the version above continues decreasing the slice until it has only one or two numbers. This new version has worst-case

complexity $O(\log |numbers|)$, since it's more efficient for some inputs. The recursive algorithm starts as follows:

1. if $numbers[start] > 0$:
 1. let $first$ be $numbers[start]$
2. otherwise if $end - start = 2$:
 1. let $first$ be $numbers[start + 1]$
3. otherwise:
 1. ...

If the slice has two numbers (step 2 is true) and the first one isn't positive (step 1 is false), then the second one must be positive (step 2.1).

13.5.2 Right number in the right place

Given $numbers$, an ascending sequence of integers without duplicates, we want to know if there's an index i such that $numbers[i] = i$. For example, $(1, 2, 3)$ doesn't have any number that matches its index, but for $(-1, 0, 2)$, number 2 is at index 2.

Like the previous problem, this one can be easily solved with a linear search, but you can do much better than that.

Exercise 13.5.3

Solve this decision problem using iterative or recursive binary search. You can outline an algorithm or write it in full, whatever you prefer. Think about the three questions at the start of this section.

Hint Answer

13.6 Divide and conquer

A decrease-and-conquer algorithm may divide the input into multiple parts, but only conquers (i.e. solves the problem for) one of them. Binary search is an example: the input sequence is divided into two halves, but only one of them is searched.

A divide-and-conquer algorithm conquers more than one part, usually all of them, and then combines their solutions. The *multiple recursion* examples in the previous chapter are divide-and-conquer algorithms.

13.6.1 Complexity

Let n be the size of the input and s be the size of the smallest input, which is necessarily a base case. Let's assume the algorithm divides the input into p parts of equal or nearly equal size. Then its complexity is defined by

- if $n = s$: $T(n) = \Theta(b)$
- if $n > s$: $T(n) = \Theta(d) + p \times T(n / p) + \Theta(c)$

where $\Theta(b)$ is the complexity of handling the base case, $\Theta(d)$ is the complexity of dividing the input and $\Theta(c)$ is the complexity of combining the subsolutions for the parts.

The expression $p \times T(n / p)$ is the time it takes to solve the p subproblems, each of size n / p .

Let's analyse the complexity of the divide-and-conquer algorithm for maximum(*numbers*, *start*, *end*), presented in the previous chapter. Remember that the input sequence isn't empty.

Maximum with slicing

The first algorithm presented was:

1. let n be $|numbers|$
2. if $n = 1$:
 1. let *solution* be head(*numbers*)
3. otherwise:
 1. let *middle* be $\text{floor}(n / 2)$
 2. let *largest left* be maximum(*numbers*[0:*middle*])
 3. let *largest right* be maximum(*numbers*[*middle*:*n*])
 4. let *solution* be $\max(\text{largest left}, \text{largest right})$

The base case has size $s = 1$ and takes constant time to process (step 2.1). Steps 3.1 to 3.3 take linear time to divide the input into $p = 2$ parts. Step 3.4 takes constant time to combine the subsolutions. We have:

- if $n = 1$: $T(n) = \Theta(1)$
- if $n > 1$: $T(n) = \Theta(n) + 2 \times T(n / 2) + \Theta(1) = 2 \times T(n / 2) + \Theta(n)$.

It has been proven that this corresponds to $T(n) = \Theta(n \times \log n)$. This is called **log-linear** complexity. It's slightly worse than linear but much better than quadratic complexity, because logarithmic run-times grow very slowly as the input size grows. In maths, the multiplication operator is omitted when that causes no confusion, so we usually write $\Theta(n \log n)$.



Info: Log-linear complexity is also called linearithmic complexity.

Maximum without slicing

The second version presented was:

1. if $start + 1 = end$:
 1. let *solution* be *numbers*[*start*]
2. otherwise:
 1. let *middle* be $start + \text{floor}((end - start) / 2)$

2. let *largest left* be maximum(*numbers*, *start*, *middle*)
3. let *largest right* be maximum(*numbers*, *middle*, *end*)
4. let *solution* be max(*largest left*, *largest right*)

The base case has size $s = 1$ and takes constant time to process (step 1.1). Steps 2.1 to 2.3 take constant time to divide the input into $p = 2$ parts. Step 2.4 takes constant time to combine the subsolutions. We have:

- if $n = 1$: $T(n) = \Theta(1)$
- if $n > 1$: $T(n) = \Theta(1) + 2 \times T(n / 2) + \Theta(1) = 2 \times T(n / 2) + \Theta(1)$.

It has been proven that this corresponds to $T(n) = \Theta(n)$.

General comments

The direct expressions for $T(n)$ remain the same for any other $p > 2$, as long as dividing and combining takes constant time. In other words, dividing into more than two parts and combining their results doesn't reduce the complexity but complicates the algorithm and increases the run-time. Therefore, most divide-and-conquer algorithms just divide the input into halves.



Note: If $T(s) = \Theta(1)$, where s is the smallest input size, and $T(n) = p \times T(n / p) + \Theta(1)$ for $n > s$ and $p > 1$, then $T(n) = \Theta(n)$. If instead $T(n) = p \times T(n / p) + \Theta(n)$, then $T(n) = \Theta(n \log n)$.

If a divide-and-conquer algorithm, like the one above, does the same steps for all inputs, i.e. there's no input for which it stops early, then the complexity obtained is both the best- and worst-case complexity. Otherwise, the recursive definition captures the worst-case complexity.

The analysis shows that it's not worth computing the maximum with a divide-and-conquer algorithm: it isn't more efficient than a much simpler iterative linear search. The next chapter presents two examples in which divide and conquer pays off.

Divide and conquer is a good approach if implemented in a parallel fashion to take advantage of multi-processor hardware. Each recursive call can be executed as a separate thread that works independently on its part of the input. The operating system allocates each thread to an available processor, reducing the time the user waits for the result, compared to executing the algorithm in one thread. Writing a multi-threaded algorithm requires special libraries or programming language constructs that are outside the scope of M269.



Info: The Operating Systems block of TM129 introduces threads.

13.7 Summary

A **divide-and-conquer** algorithm divides the input into smaller instances of the same problem, solves (conquers) each instance and combines their solutions. A **decrease-and-conquer** algorithm only solves one smaller problem instance. If the input is small enough, the solution is computed directly. Divide-and-conquer algorithms use multiple recursion; decrease-and-conquer algorithms use single recursion or iteration.

Tail-recursive algorithms can be rewritten as iterative algorithms: the loop reduces the input size or value until it's one of the base cases, which are handled after the loop.

A decrease-and-conquer approach can decrease the input by a constant amount (typically one), by a variable amount or by a constant factor (typically by halving the input).

13.7.1 Complexity

The complexity of a recursive algorithm can be defined recursively, in the form of a function (traditionally named T) on the size or value n of the input.

The base case for T is the algorithm's complexity for the base case, typically $\Theta(1)$. The recurrence relation is of the form $T(n) = T(n-1) + \Theta(\dots)$ for a decrease-by-one algorithm and $T(n) = T(n / 2) + \Theta(\dots)$ for a decrease-by-half algorithm, where $\Theta(\dots)$ is the complexity of each recursive call. The direct expression for the complexity $T(n)$ depends on the form of recurrence relation.

If $T(n) =$	then $T(n) =$	Example
$T(n - 1) + \Theta(1)$	$\Theta(n)$	length without slicing
$T(n - 1) + \Theta(n)$	$\Theta(n^2)$	length with slicing
$T(n / 2) + \Theta(1)$	$\Theta(\log n)$	binary search without slicing
$T(n / 2) + \Theta(n)$	$\Theta(n)$	binary search with slicing
$p \times T(n / p) + \Theta(1)$	$\Theta(n)$	maximum without slicing
$p \times T(n / p) + \Theta(n)$	$\Theta(n \log n)$	maximum with slicing

An algorithm with **logarithmic complexity** $\Theta(\log n)$ is very efficient: its run-time grows very slowly with input size. If the input doubles, the run-time increases by a fixed amount. We assume that exponentiation takes logarithmic time.

An algorithm with **log-linear complexity** $\Theta(n \log n)$ is less efficient than a linear algorithm but much more efficient than a quadratic algorithm.

If the exact best- or worst-case complexity isn't known, e.g. if the input is decreased each time by a variable amount, we should instead provide an upper bound (as low as we can) with **Big-Oh notation** $O(\dots)$. For example, Euclid's algorithm for the greatest common divisor of a and b has worst-case complexity $O(\log \max(a, b))$. This means that the worst-case complexity is logarithmic or better: it depends on the input values. All terminology (logarithmic, linear, exponential, etc.) applies to Big-Oh notation too.

13.7.2 Binary search

A **binary search** is a generate-and-test and a decrease-and-conquer algorithm: it generates the middle element of a sorted sequence and tests if it's the sought item. If it isn't, the search conquers either the left or the right half. Binary search keeps halving the search space after testing each candidate, until it finds a solution or the search space is empty. If the generate-and-test steps take constant time, binary search takes logarithmic time.

CHAPTER 14

SORTING

Computers spend much time sorting spreadsheet rows, sorting products by price or popularity in online stores, sorting web pages by relevance for web searches, and so on. Additionally, sorting allows other problems to be solved more easily, e.g. detecting duplicates, or faster, e.g. by using binary search.

Sorting is a fundamental computational problem for which dozens of algorithms have been invented, but only a few are used in practice. Programming languages usually have a sort function that serves most needs. It's unlikely you'll ever implement a sorting algorithm.

Nevertheless, this chapter presents several algorithms because it's instructive to see how the same problem is approached and solved in different ways. The aim is to reinforce and practise concepts, algorithmic techniques and data structures seen so far, not to learn the most efficient version of each sorting algorithm.

Sorting algorithms are mentioned in the [BBC programme](#) you watched in the first week. The first part of the programme, from 14:45 onwards, explains bubble sort and merge sort. The latter is covered in this chapter. The first 45 seconds of the second part wrap up sorting and mention pigeonhole sorting, which we will also look at.

This chapter supports these learning outcomes:

- Develop and apply algorithms and data structures to solve computational problems – this chapter applies various techniques to solve the sorting problem.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders - this chapter explains how various sorting algorithms work.
- Analyse the complexity of algorithms to support software design choices – this chapter analyses their complexity.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

14.1 Preliminaries

This section introduces the problem and the terminology used in the rest of the chapter.

14.1.1 Problem

The problem to be solved is to put the items of an input sequence in ascending order; more precisely, in non-decreasing order, because items may be duplicated. For example, if the input is $(1, 3, 2, 4, 2)$, the output is $(1, 2, 2, 3, 4)$. Sorting in descending order can be done with the same algorithm, but using the opposite comparison, so we won't consider it further. I'll use 'ascending' to mean 'non-decreasing' because the latter is a bit of a mouthful.

Imagine the items represent playing cards, with a value (ace, two, ..., ten, jack, queen, king) and a suit (clubs, diamonds, hearts, spades). Depending on the game played, we may wish to sort just by suit, just by value, by both, by colour (spades and clubs are black, hearts and diamonds are red) or even by a bespoke order. For example, depending on the game, the highest card in a suit may be the ace, the king, the seven or something else.

To allow the same items to be sorted in many different ways, we'll assume the user provides a function that computes a key for any given item. The problem then consists of putting items in ascending order of their keys. The keys must be of a comparable type, like integers or strings: otherwise it's impossible to sort them.

For example, to sort cards by value, from ace up to king, one possible function is:

$$\text{key}(\textit{value}, \textit{suit}) = \begin{cases} 1 & \text{if } \textit{value} = \text{ace} \\ 11 & \text{if } \textit{value} = \text{jack} \\ 12 & \text{if } \textit{value} = \text{queen} \\ 13 & \text{if } \textit{value} = \text{king} \\ \textit{value} & \text{otherwise} \end{cases}$$

This key function ignores the suit. If the ace is the highest instead of the lowest card, then the function must return a value higher than 13 for an ace. There are infinitely many possible functions, as long as $\text{key}(2, \textit{suit}) < \text{key}(3, \textit{suit}) < \dots < \text{key}(\text{king}, \textit{suit}) < \text{key}(\text{ace}, \textit{suit})$.

There are two versions of the sorting problem. One creates a new sorted sequence. In the following definition and the rest of this chapter, n is the length of the input sequence.

Function: create ascending sequence

Inputs: $\textit{unsorted}$, a sequence; key , a function of object to object

Preconditions: $\text{key}(a)$ and $\text{key}(b)$ are comparable for any a and b in $\textit{unsorted}$

Output: \textit{sorted} , a sequence

Postconditions:

- \textit{sorted} is a permutation of $\textit{unsorted}$
- $\text{key}(\textit{sorted}[i]) \leq \text{key}(\textit{sorted}[j])$ for every $0 \leq i < j < n$

The first postcondition states that the output has the same items as the input. The second postcondition could be stated as $\text{key}(\textit{sorted}[0]) \leq \text{key}(\textit{sorted}[1]) \leq \dots \leq \text{key}(\textit{sorted}[n-1])$.

The second version modifies the input sequence.

Operation: put in ascending order

Input/Output: $items$, a sequence

Inputs: key , a function of object to object

Preconditions: $key(a)$ and $key(b)$ are comparable for any a and b in $items$

Postconditions:

- post- $items$ is a permutation of pre- $items$
- $key(\text{post-}items[i]) \leq key(\text{post-}items[j])$ for every $0 \leq i < j < n$

The rest of this chapter assumes that sequences are represented as arrays, so that algorithms can efficiently access any item in the sequence.

14.1.2 Problem instances

To test the sorting algorithms to be presented, I'll use playing cards, each represented by a string of length 2, e.g. 'AS' (ace of spades), '7H' (seven of hearts) and 'TD' (ten of diamonds).

Many other representations of cards are possible. Some are easier for a user to understand; others make key functions easier to implement. I chose this one because it makes tests quick to type and easy to understand. It's up to the key function to transform a user-friendly representation into a sortable key. I define three key functions:

- A key function that returns the value of a card as an integer from 1 to 13 allows us to sort the cards by ascending value.
- A key function that returns the suit of a card allows us to sort cards alphabetically by suit: clubs, diamonds, hearts, spades.
- A key function that returns the suit–value pair allows us to sort the cards first alphabetically by suit and within the same suit by ascending value, due to the *lexicographic comparison* of pairs.

```
[1]: # this code is also in m269_sorting.py
```

```
def suit(card: str) -> str:
    """Return the second character of the card.

    Preconditions: card has two characters;
    the first is 'A', '2' to '9', 'T', 'J', 'Q' or 'K'
    the second is 'C', 'D', 'H' or 'S'
    """
    return card[1]
```

```
VALUES = "A23456789TJQK"
```

```
def value(card: str) -> int:
```

(continues on next page)

(continued from previous page)

```
"""Return the value of the card.

Preconditions: as for function 'suit'
Postconditions: the output is 1 to 13 respectively for
'A', '2' to '9', 'T', 'J', 'Q', 'K'
"""

for index in range(len(VALUES)):
    if VALUES[index] == card[0]:
        return index + 1 # return 1-13, not 0-12

def suit_value(card: str) -> tuple:
    """Return a tuple with the suit and value of the card.

    Preconditions: as for function 'suit'
    """

    return (suit(card), value(card))
```

```
[2]: suit_value("TD")
[2]: ('D', 10)
```

I can now write a few tests, using Python lists for the sequences. These tests will only work for the first version of the problem, where the sorting function returns a sorted sequence.

```
[3]: # this code is also in m269_sorting.py

UP_DOWN = ["AS", "3H", "QD", "KC"] # ascending values, descending
→suits
SAME_VALUE = ["TD", "TS", "TH", "TC"]

sorting_tests = [
    # case,           unsorted,          key,   sorted
    ('empty list', [],            suit_value, []),
    ('1 card',     ['AS'],        suit_value, ['AS']),
    ('same cards', ['6D', '6D'], suit_value, ['6D', '6D']),
    ('3 cards',    ['JC', '8D', 'TS'], value, ['8D', 'TS', 'JC']),
    ('values up',   UP_DOWN,      value, UP_DOWN),
    ('suits down', UP_DOWN,      suit,  ['KC', 'QD', '3H', 'AS']),
    ('same value',  SAME_VALUE,  suit_value, ['TC', 'TD', 'TH', 'TS']),
]
```

To check the test table, I must tell `check_tests` that the 'key' argument is a function. In Python, functions are objects of type `Callable`, defined in the `typing` module.

```
[4]: from typing import Callable
from algoesup import check_tests
```

(continues on next page)

(continued from previous page)

```
check_tests(sorting_tests, [list, Callable, list])
OK: the test table passed the automatic checks.
```

To measure the performance of sorting algorithms we need long sequences, but there are only 52 cards. I'll use long sequences of integers. Integers can be compared, so a key function that returns the item itself suffices. A function that returns its input is called the identity function.

[5]: # this code is also in m269_sorting.py

```
def identity(item: object) -> object:
    """Return the item, i.e. the key is the whole item."""
    return item
```

14.1.3 Algorithms

An algorithm is **in-place** if it doesn't use any additional memory, other than the call stack and a fixed number of local variables for individual items in the sequence, for indices, Booleans, etc. For example, finding a pair of items that add up to a given amount can be done in-place with an exhaustive search (nested loop) for all pairs, or not in-place, with an additional map as in an *earlier exercise*.

Sorting algorithms that return a new sorted sequence aren't in-place. Those that modify the input sequence are usually in-place, but if an algorithm would create a temporary new sorted sequence and then copy the items to the input sequence, it wouldn't be in-place.

14.1.4 Sorting in Python

You've seen in [Chapter 4](#) that Python has a `sorted` function which returns a sorted list from a given sequence and a list method `sort` which modifies the list. The method sorts the input sequence in-place; the function doesn't. Both can take an argument indicating which key function to use.

[6]:

```
items = ["2S", "AS", "2D", "AD"]
items.sort(key=suit_value)
items
```

[6]:

```
['AD', '2D', 'AS', '2S']
```

Like for the `reverse` parameter, omitting the parameter name leads to an error.

[7]:

```
items.sort(suit_value)
```

TypeError

Traceback (most recent)

(continues on next page)

(continued from previous page)

```
→call last)
Cell In[7], line 1
----> 1 items.sort(suit_value)

TypeError: sort() takes no positional arguments
```

Python uses the Powersort algorithm, an improvement over Timsort, which in turn is derived from the insertion sort and merge sort algorithms explained in later sections. Powersort and Timsort are in-place algorithms with linear complexity in the best case and log-linear complexity in the worst case. You're unlikely to need another algorithm for sorting arrays in memory. I won't explain Powersort, Timsort or how to sort data that doesn't fit in memory. This chapter isn't about the best way of sorting; instead it presents algorithms that illustrate previous concepts and techniques.

14.2 Bogosort

Every proper textbook includes some daft algorithms to show when *not* to use a particular technique or data structure. Bogosort is such an algorithm.

The postconditions of the sorting problem state that the output sequence is a permutation of the input sequence that is in ascending order. Bogosort approaches the sorting problem as a search problem: find a permutation of the input sequence that is in ascending order. Since the keys are pairwise comparable, at least one permutation is in ascending order: the search is guaranteed to be successful.

The exhaustive search algorithm simply generates each permutation and tests if it's in ascending order with an auxiliary function.

1. for each permutation p of *unsorted*:
 1. if `isAscending(p)`:
 1. let *sorted* be p
 2. stop

You wrote a recursive definition of the auxiliary test function *earlier*. I'm sure you can write an iterative algorithm too.

In the best-case scenario for the test, the first two items of a permutation are in descending order and so the test returns false in constant time. In the worst-case scenario, the permutation is in ascending order and the test takes linear time $\Theta(n)$ to return true.

What are the best- and worst-case complexities of bogosort? Assume that the first permutation generated is the input sequence.

In the best case, the input sequence is already sorted, and so the algorithm only generates and tests a single candidate in linear time. In the worst case, the algorithm generates and tests all permutations in $\Theta(n! \times n)$ time.

The aim of this section is to illustrate, once more, that many problems can be cast as search problems, but that exhaustive search isn't always appropriate. Given the appalling complexity of bogosort, I won't implement it. That's why we do the complexity analysis before coding.

If you think bogosort is bad, here's a worse version: keep generating random permutations and check if they're sorted. Deterministic bogosort, the first version presented, generates the candidate permutations systematically and eventually terminates. Randomised bogosort may continue forever ...

14.3 Insertion sort

Let's apply decrease and conquer to sorting, instead of exhaustive search.

14.3.1 Recursive version

I start with the usual questions.

1. What are the smallest possible unsorted sequence and its sorted counterpart? The unsorted and sorted sequences are empty.
2. If the tail were already sorted, how could we sort the whole sequence? We simply insert the head in the appropriate place.

This is the core idea of **insertion sort**: the next item to be processed is inserted into an already sorted part of the sequence. After processing all items, the sequence is sorted.

This is one way players sort their cards: with one hand they pick up the cards they were dealt, one by one, inserting each one in its place in a fan of cards held in their other hand.

The algorithm for function `insertion_sorted(unsorted, key)` is:

1. if *unsorted* is empty:
 1. let *sorted* be ()
2. otherwise:
 1. let *subsolution* be `insertion_sorted(tail(unsorted), key)`
 2. let *sorted* be `insert(head(unsorted), subsolution, key)`

Inserting an item into a sorted sequence (step 2.2) also requires the key function. The algorithm for function `insert(item, sorted, key)` has to compare the keys of the item to be inserted and of the head, to know if the item is to be inserted before or after the head. It's another decrease-and-conquer algorithm. I call the output variable *inserted*.

1. if *sorted* is empty:
 1. let *inserted* be (*item*)
2. otherwise if $\text{key}(\text{item}) \leq \text{key}(\text{head}(\text{sorted}))$:
 1. let *inserted* be `prepend(item, sorted)`
3. otherwise:

1. let *subsolution* be insert(*item*, tail(*sorted*), *key*)
2. let *inserted* be prepend(head(*sorted*), *subsolution*)

This version of insertion sort isn't tail recursive. It exhausts the call stack when sorting a sequence with thousands of items. We need an iterative version.

14.3.2 Iterative version

Here's a version that can be implemented in-place, on the input array.

Continuing with the card game analogy, the player fans all the dealt cards in one hand and mentally divides them in two parts: the sorted cards are on the left and the unsorted cards are on the right. Initially, the sorted part is the first card and the unsorted part has all other ones. At each step, the player uses their free hand to pick the left-most unsorted card and put it in its correct place in the sorted part. Each step grows the sorted part by one card and shrinks the unsorted part by one card. The sorting ends when the unsorted part is empty.

The next table shows step by step how in-place insertion sort alphabetically sorts a sequence of seven letters. I use monospaced font to highlight the already sorted left part of the sequence.

0	1	2	3	4	5	6
S	O	R	T	I	N	G
O	S	R	T	I	N	G
O	R	S	T	I	N	G
O	R	S	T	I	N	G
I	O	R	S	T	N	G
I	N	O	R	S	T	G
G	I	N	O	R	S	T

To insert each unsorted item in the sorted part, the algorithm shifts right all sorted items that are larger than the unsorted item. This overwrites the unsorted item but makes place to put it in the sorted part. This [visualisation](#) shows the shifting in more detail.

I proceed directly to the code. As shown in the visualisation, insertion sort goes through the sorted items from right to left and shifts each one to the right if it's larger than the unsorted item.

Remember that our sorting algorithms take a *key function* as a second argument and that functions are objects of type `Callable`.

```
[1]: from typing import Callable

def insertion_sort(items: list, key: Callable) -> None:
    """Sort the items in-place, with keys in non-decreasing order.

    Preconditions: for any indices i and j,
    key(items[i]) and key(items[j]) are comparable
    """
    pass
```

(continues on next page)

(continued from previous page)

```

# go through all items in the unsorted part
for first_unsorted in range(1, len(items)):
    to_sort = items[first_unsorted]
    # apply the key function given as input
    the_key = key(to_sort)
    # to start, index where to put item is index where it is now
    index = first_unsorted
    # for each sorted item larger than the one to sort
    while index > 0 and key(items[index - 1]) > the_key:
        # copy it to the right, i.e. one position up
        items[index] = items[index - 1]
        # and proceed with the next sorted item on the left
        index = index - 1
    # sorted item on the left isn't larger: we found the index
    items[index] = to_sort # put the unsorted item there

```

There are *two versions* of the sorting problem: one modifies the input, the other returns a sorted copy. I follow the same naming convention as Python: `sort` for the in-place version and `sorted` for the other.

To use the `test` function, I need the sorting function to return an output, so I will implement the `sorted` version that makes a copy of the input list, using the `list` constructor: `copy = list(original)`. Note that `copy = original` would create a new reference to the *same* list and hence modify the original input.

```
[2]: from algoesup import test

%run -i ../m269_sorting

def insertion_sorted(unsorted: list, key: Callable) -> list:
    """Return a permutation with keys in non-decreasing order.

    Preconditions: for any indices i and j,
    key(unsorted[i]) and key(unsorted[j]) are comparable
    """
    result = list(unsorted) # make a copy
    insertion_sort(result, key)
    return result

test(insertion_sorted, sorting_tests)

Testing insertion_sorted...
Tests finished: 7 passed (100%), 0 failed.
```

Remember that our *sorting test table* indicates which key function (`value`, `suit` or `suit_value`) each test uses.

14.3.3 Complexity

For this and the following algorithms, we assume that the key function takes constant time and so does comparing two keys.

The algorithm does the least work when the while-loop never executes. This happens if the condition is always false: the key of the left neighbour of the unsorted item has a lower or equal key. In that case, the `index` variable doesn't change and the unsorted item remains in its place, becoming the new right-most sorted item. What's the best-case complexity of insertion sort?

The for-loop does $n - 1$ iterations, each only doing constant-time operations, because the while-loop is skipped. The complexity is $\Theta(n)$.

The algorithm does the most work if, for each unsorted item, *all* the sorted items are shifted. The first unsorted item requires one shift because the sorted part has one item. The second unsorted item requires two shifts because the sorted part has now two items. In total, the number of shifts is $1 + 2 + \dots + n-1 = ((n-1)^2 + (n-1)) / 2$ by using the formula seen in the [previous chapter](#). Each shift takes constant time, so the worst-case complexity is $\Theta(n^2)$. Can you think of a scenario with this worst-case complexity?

If inserting an unsorted item shifts all sorted items, then the unsorted item must be smaller than all of them and go to index zero. If each unsorted item triggers shifting, then it must be smaller than the previous unsorted item. A worst-case scenario is therefore the sequence being in descending order, or more generally, in reverse sorted order.

14.3.4 Performance

Let's check the best-case complexity by generating ascending integer sequences, always doubling the length. As previously advised, I don't start with very short sequences, but not too long ones either because the worst-case is quadratic.

```
[3]: %run -i ../m269_sorting

for doubling in range(5):
    # generate lists of length 100, 200, 400, 800, 1600
    # each list has the integers from 0 to length - 1
    items = list(range(100 * 2**doubling))
    %timeit -r 5 insertion_sorted(items, identity)

6.95 µs ± 9.73 ns per loop (mean ± std. dev. of 5 runs, 100,000 loops
 ↴ each)
13.7 µs ± 9.58 ns per loop (mean ± std. dev. of 5 runs, 100,000 loops
 ↴ each)
30.4 µs ± 162 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴ each)
65 µs ± 43.8 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴ each)
135 µs ± 92.1 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴ each)
```

Let's check the worst-case complexity with descending sequences.

```
[4]: for doubling in range(5):
    items = list(range(100 * 2**doubling, -1, -1))
    %timeit -r 5 insertion_sorted(items, identity)

280 µs ± 147 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↴each)
1.1 ms ± 806 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↴each)
4.74 ms ± 11.2 µs per loop (mean ± std. dev. of 5 runs, 100 loops
 ↴each)
21.4 ms ± 16.1 µs per loop (mean ± std. dev. of 5 runs, 10 loops
 ↴each)
92.5 ms ± 72.6 µs per loop (mean ± std. dev. of 5 runs, 10 loops
 ↴each)
```

Do the run-times confirm the best- and worst-case complexities?

For ascending sequences, the run-time doubles as the length doubles, thus confirming the linear complexity. For descending sequences, the run-time roughly quadruples as the length doubles, thus confirming the quadratic complexity.

Exercise 14.3.1

The above code isn't just measuring the run-time of insertion sort: it's also measuring the time to copy the input list. Wouldn't it be better to directly call the in-place `insertion_sort` instead of `insertion_sorted`?

Hint Answer

14.4 Selection sort

Decrease-by-one algorithms split the input collection into one item and the rest, recursively process the rest and then, if necessary, combine the subsolution with the removed item.

Non-empty sequences are easily split into their head and tail, but considering a different split leads to a new sorting algorithm.

14.4.1 Recursive version

To split a sequence differently, we ask ourselves what item should be removed so that the combination step becomes easier.

For a sorting algorithm, a possible answer is: the item with the smallest key. Once we sorted the remaining items, the combination step just prepends the smallest item to put it at the start of the sorted sequence. This is called **selection sort** because at each step we select the smallest of the still-unsorted items.

Instead of extracting the head and tail of a sequence, we need to extract the minimum and the rest, without the minimum. Both auxiliary functions must use the key function to do their job. Let's call them 'min' and 'without_min'. The algorithm for selection_sorted(*unsorted*, *key*) is:

1. if *unsorted* is empty:
 1. let *sorted* be ()
2. otherwise:
 1. let *decreased* be without_min(*unsorted*, *key*)
 2. let *subsolution* be selection_sorted(*decreased*, *key*)
 3. let *sorted* be prepend(min(*unsorted*, *key*), *subsolution*)

This isn't a tail-recursive algorithm, so I'll skip the algorithms for the auxiliary functions and proceed to an iterative version.

14.4.2 Iterative version

Selection sort is another common way to sort cards. We pick up all dealt cards in one go and fan them in one hand. With the other hand, we pick the lowest card and put it in the left-most position of the fan. Then we choose the next lowest card and insert it in the second position. We repeat this until all cards are sorted.

Like for insertion sort, the fan is divided into a left sorted part that grows and a right unsorted part that shrinks. In selection sort, we move the smallest card in the unsorted part to the end of the sorted part.

Both algorithms do two things:

1. choose the next item to process from the unsorted part
2. put it in its correct place in the sorted part.

Insertion sort does less work in step 1 (take the next unsorted item) and more work in step 2 (search for the place where to insert the item). Selection sort does more work in step 1 (search for the smallest item) and less in step 2 (append the item to the sorted part).

In a way, insertion and selection sort are the inverse of each other. Insertion sort knows where the next item to sort is but doesn't know where it will end up, whereas selection sort doesn't know where the next item to sort is but knows where it needs to end up.

The in-place version of selection sort swaps the smallest unsorted item with the left-most unsorted item. This puts the smallest item in its place, without shifting any items.

The next table again shows the sorted part in monospaced font. The first letter S is swapped with G, the alphabetically first letter in the unsorted part. Then the second letter O is swapped with I, the alphabetically first letter in the remaining unsorted part, and so on.

0	1	2	3	4	5	6
S	O	R	T	I	N	G
G	O	R	T	I	N	S
G	I	R	T	O	N	S
G	I	N	T	O	R	S
G	I	N	O	T	R	S
G	I	N	O	R	T	S
G	I	N	O	R	S	T

Insertion sort starts with a sorted part with one item and stops when the unsorted part is empty. Selection sort starts with an empty sorted part, and stops when the unsorted part has one item. Because selection sort moves in each step the smallest item to the sorted part, all unsorted items are larger than all sorted items. After sorting $n - 1$ items, the remaining item is the largest one and is already in the last position.

Selection sort minimises the number of swaps: it does at most $n - 1$ swaps because each item gets into its place with a single swap. The in-place algorithm is:

1. for each *first unsorted* from 0 to $n - 2$:
 1. let *smallest* be the index of the item with smallest key in *items[first unsorted:n]*
 2. swap *items[first unsorted]* and *items[smallest]*

Exercise 14.4.1

What's the complexity of in-place selection sort?

Answer

14.4.3 Code

The implementation of the algorithm consists of a linear search for the item with the smallest key, from the first unsorted item onwards, followed by a swap with the first unsorted item.

```
[1]: from typing import Callable

def selection_sort(items: list, key: Callable) -> None:
    """Sort the items in-place, with keys in non-decreasing order.

    Preconditions: for any indices i and j,
    key(items[i]) and key(items[j]) are comparable
    """
    for first_unsorted in range(0, len(items) - 1):
        smallest = first_unsorted
        for index in range(smallest + 1, len(items)):
            if key(items[index]) < key(items[smallest]):
```

(continues on next page)

(continued from previous page)

```
        smallest = index
    unsorted_item = items[first_unsorted]
    items[first_unsorted] = items[smallest]
    items[smallest] = unsorted_item
```

Like for insertion sort, we need a version that isn't in-place for testing.

```
[2]: from algoesup import test
```

```
%run -i ../m269_sorting
```

```
def selection_sorted(unsorted: list, key: Callable) -> list:
    """Return a permutation with keys in non-decreasing order.
```

*Preconditions: for any indices i and j,
key(unsorted[i]) and key(unsorted[j]) are comparable*

```
result = list(unsorted) # make a copy
selection_sort(result, key)
return result
```

```
test(selection_sorted, sorting_tests)
```

```
Testing selection_sorted...
```

```
Tests finished: 7 passed (100%), 0 failed.
```

Let's check the complexity is always quadratic, even when the sequence is sorted.

```
[3]: for doubling in range(5):
    items = list(range(100 * 2**doubling)) # sorted order
    %timeit -r 5 selection_sorted(items, identity)
```

```
230 µs ± 292 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
→ each)
```

```
906 µs ± 398 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
→ each)
```

```
4.12 ms ± 10.6 µs per loop (mean ± std. dev. of 5 runs, 100 loops
→ each)
```

```
17.5 ms ± 56.9 µs per loop (mean ± std. dev. of 5 runs, 100 loops
→ each)
```

```
70.6 ms ± 121 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
```

```
[4]: for doubling in range(5):
    items = list(range(100 * 2**doubling, -1, -1)) # reverse order
    %timeit -r 5 selection_sorted(items, identity)
```

```

240 µs ± 184 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↵each)
938 µs ± 1.06 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↵each)
4.26 ms ± 10 µs per loop (mean ± std. dev. of 5 runs, 100 loops each)
18.2 ms ± 53.8 µs per loop (mean ± std. dev. of 5 runs, 100 loops
 ↵each)
72.9 ms ± 63.2 µs per loop (mean ± std. dev. of 5 runs, 10 loops
 ↵each)
    
```

The run-times are similar for ascending and descending sequences, and quadruple as the length doubles. This confirms the constantly quadratic complexity. (I like alliterations a lot.)

14.4.4 Select largest

The selection sort algorithm swaps the *smallest* and *first* unsorted items. It's also possible to swap the *largest* and *last* unsorted items, i.e. to move the largest item to the end of the unsorted part. In this 'mirror' version, the sorted part is on the right and the unsorted part is on the left. [This visualisation](#) shows the approach.

Exercise 14.4.2

Redo the example with the word SORTING for this 'select largest' version. Don't write a table as I did: just write the sequence of letters as it changes from SORTING to GINORST, one sequence per line, with a hyphen to separate the right sorted part from the left unsorted part.

SORTING—

...

G-INORST

Hint Answer

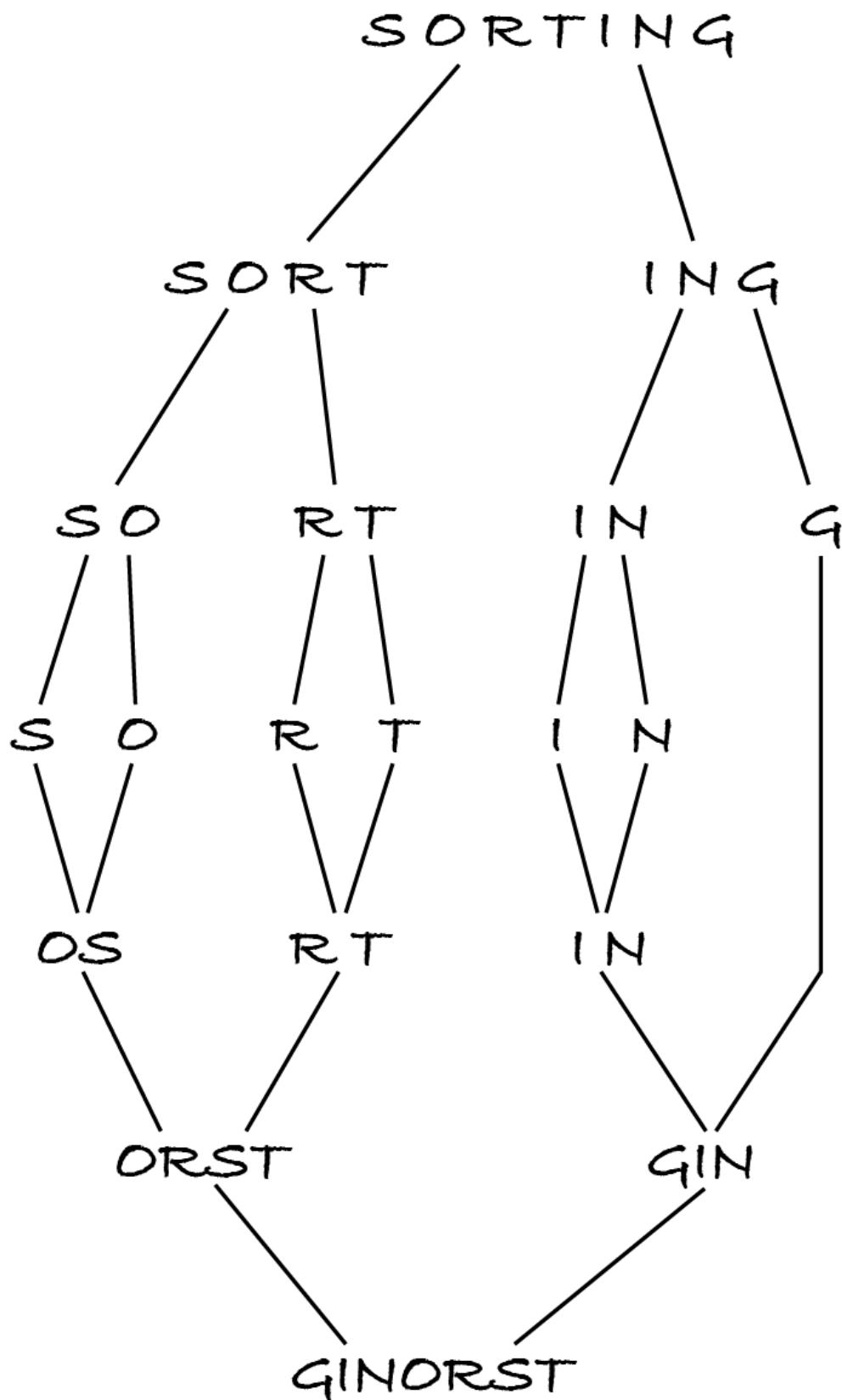
14.5 Merge sort

Let's now look at a divide-and-conquer approach to sorting.

14.5.1 Algorithm

Merge sort divides the sequence into two halves, sorts each one recursively, and merges them by taking the smallest item of each half until both are empty. The base cases are sequences of length 0 or 1, as they can't be split into smaller sequences. When the length is odd, it doesn't matter which half has one element more. The next figure shows the process applied to our familiar example.

Figure 14.5.1



Here's the recursive algorithm for mergesort(*unsorted*, *key*):

1. if $n < 2$:
 1. let *sorted* be *unsorted*
2. otherwise:
 1. let *middle* be $\text{floor}(n / 2)$
 2. let *left sorted* be mergesort(*unsorted*[0:*middle*], *key*)
 3. let *right sorted* be mergesort(*unsorted*[*middle*:*n*], *key*)
 4. let *sorted* be merge(*left sorted*, *right sorted*, *key*)

The previous M269 team produced a [visualisation](#) of step 2.4, so that you can see it in more detail. The code shown in the visualisation doesn't use a key function. Here's the algorithm for merge(*left*, *right*, *key*) with the precondition that both input sequences are sorted.

1. let *left index* be 0
2. let *right index* be 0
3. let *sorted* be the empty sequence
4. while $\text{left index} < |\text{left sorted}|$ and $\text{right index} < |\text{right sorted}|$:
 1. let *left item* be *left sorted*[*left index*]
 2. let *right item* be *right sorted*[*right index*]
 3. if $\text{key}(\text{left item}) < \text{key}(\text{right item})$:
 1. append *left item* to *sorted*
 2. let *left index* be *left index* + 1
 4. otherwise:
 1. append *right item* to *sorted*
 2. let *right index* be *right index* + 1
5. for *index* from *left index* to $|\text{left sorted}| - 1$:
 1. append *left sorted*[*index*] to *sorted*
6. for *index* from *right index* to $|\text{right sorted}| - 1$:
 1. append *right sorted*[*index*] to *sorted*

Step 4 and its sub-steps merge both sequences until one is copied to the output. The unprocessed items in the other half are then copied by either step 5 or step 6. Only one of those for-loops will execute.

14.5.2 Complexity

Let's first obtain the complexity in a visual way. In the above figure, each level splits sequences in half and a later corresponding level merges the sorted halves. Each splitting and merging corresponds to one recursive call. Since the input is always split in half, there are $\log_2 n$ recursive calls until the sequences have length 1 and start being merged. In the example, $n = 7$, so there are 3 recursive call levels.

Each level processes the whole input, of length n , albeit split in substrings. For example, the second level processes 4 sequences of one or two characters each, while the third level processes seven sequences of one character each. Processing each character takes constant time, both when it's being copied from the input to an unsorted half and when it's copied from a sorted half to the output. The complexity is therefore $\log n \times n \times \Theta(1) = \Theta(n \log n)$.

Informal reasoning can sometimes have subtle flaws, so it's safest to systematically define the complexity recursively, following the algorithm.

Exercise 14.5.1

Write the recursive definition of T and confirm that merge sort has log-linear complexity.

- if $n < 2$: $T(n) = \dots$
- if $n \geq 2$: $T(n) = \dots$

Hint Answer

Merge sort has two advantages over insertion and selection sort: it has better than quadratic complexity for unsorted input sequences and, being a divide-and-conquer algorithm, can be implemented in parallel.

14.5.3 Code and performance

```
[1]: from typing import Callable
from algoesup import test

%run -i ../m269_sorting

def merge(left: list, right: list, key: Callable) -> list:
    """Merge both lists into one with keys in non-decreasing order.

    Preconditions: left and right have keys in non-decreasing order
    """
    left_index = 0
    right_index = 0
    result = []
    while left_index < len(left) and right_index < len(right):
        left_item = left[left_index]
        right_item = right[right_index]
        if key(left_item) < key(right_item):
```

(continues on next page)

(continued from previous page)

```

        result.append(left_item)
        left_index = left_index + 1
    else:
        result.append(right_item)
        right_index = right_index + 1
    for index in range(left_index, len(left)):
        result.append(left[index])
    for index in range(right_index, len(right)):
        result.append(right[index])
    return result

def merge_sorted(unsorted: list, key: Callable) -> list:
    """Return a permutation with keys in non-decreasing order.

    Preconditions: for any indices i and j,
    key(unsorted[i]) and key(unsorted[j]) are comparable
    """
    n = len(unsorted)
    if n < 2:
        return unsorted
    else:
        middle = n // 2
        left_sorted = merge_sorted(unsorted[:middle], key)
        right_sorted = merge_sorted(unsorted[middle:], key)
        return merge(left_sorted, right_sorted, key)

test(merge_sorted, sorting_tests)
Testing merge_sorted...
Tests finished: 7 passed (100%), 0 failed.

```

Remember that the start and end indices of a slice *can be omitted*.

Since the complexity is always the same, I measure the run-time for sequences that are easy to generate.

```
[2]: for doubling in range(5):
    items = [0] * 100 * 2**doubling # 100, 200, 400, ... zeros
    %timeit -r 5 merge_sorted(items, identity)

73.8 µs ± 86.7 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops±
 ↪each)
160 µs ± 606 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops±
 ↪each)
340 µs ± 262 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops±
 ↪each)
```

(continues on next page)

(continued from previous page)

```
730 µs ± 1.33 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops→each)
1.58 ms ± 5.65 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops→each)
```

The run-times more than double but don't quadruple when the input size doubles. This confirms the complexity is between linear and quadratic.

14.6 Quicksort

Like merge sort, **quicksort** divides the input sequence in two partitions, recursively sorts each partition and then puts them together. Whereas merge sort divides the sequence by position, in two halves, quicksort divides the items by key: those with smaller keys go into one partition, those with larger keys go in the other. After the partitions are sorted, they just have to be concatenated, which is much simpler than merging.

Insertion sort did less work than selection sort when splitting the input but more when combining the subsolution with the removed item. Likewise, merge sort does less work than quicksort when splitting but more work when combining the subsolutions.



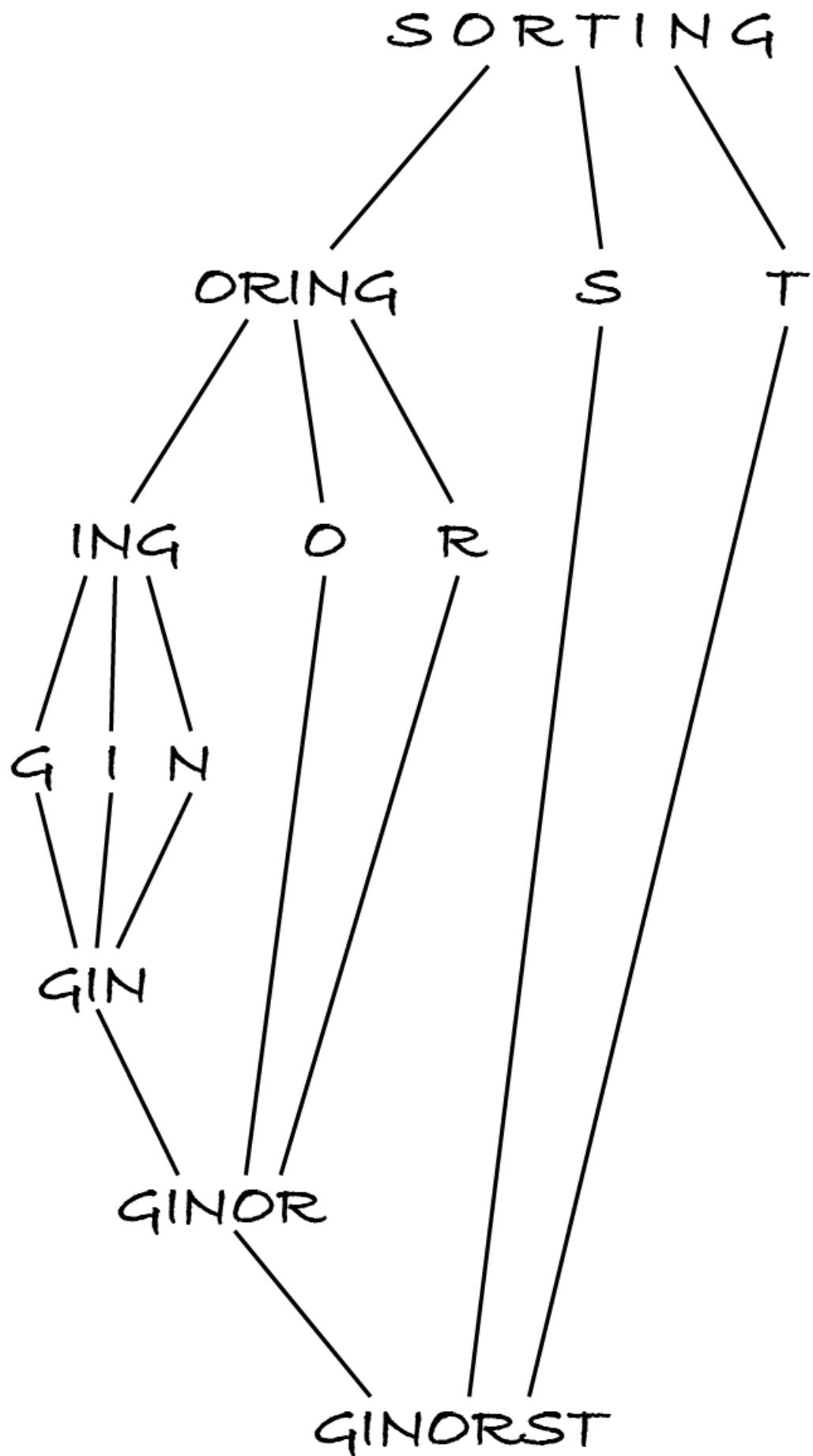
Note: You can design different decrease-and-conquer or divide-and-conquer algorithms by making different phases of the approach simpler.

14.6.1 Algorithm

Quicksort starts by choosing one item as the **pivot**, then splits the other items into those smaller and those larger than the pivot. After each partition is sorted, they're put together: first the smaller items, then the pivot and finally the larger items.

Here's how quicksort processes our example, with the pivot being the first item:

Figure 14.6.1



In the first step, the pivot is S, the letters that come before S are ORING and the only letter that comes after is T. After sorting ORING into GINOR, the S and T are concatenated back in that order, resulting in GINORST.

Here's a recursive quicksort algorithm. It uses an auxiliary function that partitions the unsorted sequence into three sequences: the smaller items, a sequence of length 1 with the pivot and the sequence of larger items.

1. if $n < 2$:
 1. let *sorted* be *unsorted*
2. otherwise:
 1. let $(smaller, pivot, larger)$ be partition(*unsorted*)
 2. let *sorted* be quicksort(*smaller*, *key*) concatenated with *pivot* and quicksort(*larger*, *key*)

Step 2.1 is an abuse of the assignment notation to make the algorithm more readable.

The partition function simply chooses the first item as the pivot and adds the other items to either partition, depending on how they compare to the pivot.

1. let *smaller* be ()
2. let *larger* be ()
3. let *pivot* be *unsorted*[0]
4. for each *index* from 1 to $n - 1$:
 1. let *item* be *unsorted*[*index*]
 2. if $key(item) < key(pivot)$:
 1. append *item* to *smaller*
 3. otherwise:
 1. append *item* to *larger*
5. let *output* be $(smaller, (pivot), larger)$

Note that the final step doesn't return the pivot but a sequence with it, so that the concatenation operation can be applied in step 2.2 of quicksort.

14.6.2 Complexity

Each recursive call goes through its input sequence twice: first to partition it and then to concatenate the partitions and the pivot. The recursive complexity definition is:

- if $n < 2$: $T(n) = \Theta(1)$
- if $n \geq 2$: $T(n) = \Theta(n) + T(|smaller|) + T(|larger|) + \Theta(n) = T(|smaller|) + T(|larger|) + \Theta(n)$.

If the input is sorted, then the pivot (the first item) is the smallest one. So all other items are put in partition *larger* and partition *smaller* is empty. The recurrence relation becomes:

$T(n) = T(0) + T(n - 1) + \Theta(n) = \Theta(1) + T(n - 1) + \Theta(n) = T(n - 1) + \Theta(n)$.

We've seen before that this resolves to $T(n) = \Theta(n^2)$.

In the best-case scenario, the pivot is the middle value and quicksort halves the sequence, like merge sort. The recurrence becomes:

$T(n) = T(n / 2) + T(n / 2) + \Theta(n) = 2 \times T(n / 2) + \Theta(n)$.

This resolves to $T(n) = \Theta(n \log n)$.

It has been proven that the average complexity of quicksort, when items are in random order, is also log-linear.

Exercise 14.6.1

What's the complexity of quicksort if the input is in reverse order?

Hint Answer

As the analysis shows, the choice of pivot is crucial to achieve log-linear complexity. One common approach is to choose a random item. Another way is to pick the median of the first, middle and last items. Unless we're unlucky and those three items have duplicate keys, this guarantees the pivot has neither the lowest nor the highest key in the sequence, which would lead to quadratic complexity.

14.6.3 Code and performance

The code mainly follows the algorithm with two small changes. First, the pivot's key is computed only once. Second, the partitioning algorithm isn't in a separate function. This makes the code shorter and easier to follow, in my opinion.

```
[1]: from typing import Callable
from algoesup import test

%run -i ../m269_sorting

def quick_sorted(unsorted: list, key: Callable) -> list:
    """Return a permutation with keys in non-decreasing order.

    Preconditions: for any indices i and j,
    key(unsorted[i]) and key(unsorted[j]) are comparable
    """
    # base case: sequences with 0 or 1 items are sorted
    if len(unsorted) < 2:
        return unsorted
    else:
        # divide the input: select the pivot and create the
        # partitions
        smaller = []
        larger = []
        pivot_index = len(unsorted) // 2
        pivot_value = key(unsorted[pivot_index])
        for item in unsorted:
            if key(item) < pivot_value:
                smaller.append(item)
            else:
                larger.append(item)
        return quick_sorted(smaller, key) + [pivot_value] + quick_sorted(larger, key)
```

(continues on next page)

(continued from previous page)

```
larger = []
pivot = unsorted[0]
pivot_key = key(pivot)
for index in range(1, len(unsorted)):
    item = unsorted[index]
    if key(item) < pivot_key:
        smaller.append(item)
    else:
        larger.append(item)
# recur into the partitions and combine the results
return quick_sorted(smaller, key) + [pivot] + quick_
sorted(larger, key)

test(quick_sorted, sorting_tests)

Testing quick_sorted...
Tests finished: 7 passed (100%), 0 failed.
```

Let's confirm that sorting an ascending sequence takes quadratic time:

```
[2]: for doubling in range(5):
    items = list(range(100 * 2**doubling))
    %timeit -r 5 quick_sorted(items, identity)

256 µs ± 865 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops_
↪each)
951 µs ± 2.99 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops_
↪each)
3.71 ms ± 2.53 µs per loop (mean ± std. dev. of 5 runs, 100 loops_
↪each)
15.2 ms ± 8.48 µs per loop (mean ± std. dev. of 5 runs, 100 loops_
↪each)
61.9 ms ± 30 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
```

To observe the log-linear complexity, I use a Python function to shuffle the items to put them in random order:

```
[3]: from random import shuffle

for doubling in range(5):
    items = list(range(100 * 2**doubling))
    shuffle(items)
    %timeit -r 5 quick_sorted(items, identity)

47.8 µs ± 109 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
↪each)
111 µs ± 114 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops_
↪each)
```

(continues on next page)

(continued from previous page)

```

↪each)
253 µs ± 646 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops)
↪each)
588 µs ± 2.28 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops)
↪each)
1.24 ms ± 996 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops)
↪each)
    
```

14.6.4 In-place version

Quicksort is usually implemented in-place, swapping smaller and larger items so that the smaller items end up in the left-hand part of the sequence and the larger items in the right-hand part, with the pivot between them. Once each part is sorted, no concatenation is necessary. A visualisation explaining the in-place algorithm is [here](#).

In-place quicksort uses less memory and is much faster than the version above because it doesn't create and concatenate sequences. It nevertheless has the same best-, average- and worst-case complexities.



Info: Many programming languages used some variant of quicksort before Timsort was invented.

14.7 Quicksort variants

This section presents two variations on quicksort to further reinforce the divide-and-conquer approach and its relation to decrease and conquer.

14.7.1 Three-way quicksort

Divide and conquer doesn't have to be in halves. We can partition the *unsorted* sequence in three, with the items smaller than, equal to and larger than the pivot. Items with the same key as the pivot don't have to be further sorted.

The main quicksort algorithm stays the same, because it already divides the input in three sequences and recurs into two of them.

1. if $n < 2$:

 1. let *sorted* be *unsorted*

2. otherwise:
 1. let $(smaller, pivot, larger)$ be *partition(unsorted)*
 2. let *sorted* be *quicksort(smaller, key)* concatenated with *pivot* and *quicksort(larger, key)*

The partition function does change slightly: the middle sequence is no longer one item (the pivot), but is all items with the same key as the pivot.

I take the opportunity to choose a random pivot to reduce the chance of quadratic complexity for already sorted inputs.

1. let *smaller* be the empty sequence
2. let *equal* be the empty sequence
3. let *larger* be the empty sequence
4. let *pivot* be a random element of *unsorted*
5. for each *item* in *unsorted*:
 1. if $\text{key}(\text{item}) < \text{key}(\text{pivot})$:
 1. append *item* to *smaller*
 2. otherwise if $\text{key}(\text{item}) = \text{key}(\text{pivot})$:
 1. append *item* to *equal*
 3. otherwise
 1. append *item* to *larger*
6. let *output* be (*smaller*, *equal*, *larger*)

Exercise 14.7.1

If all items in the input sequence have the same key, what's the complexity of

- ‘normal’ quicksort, i.e. three-way quicksort without steps 5.2 and 5.2.1?
- three-way quicksort?

Hint Answer

Three-way quicksort still has quadratic worst-case complexity if each chosen pivot has the lowest or highest key. However, it's unlikely that every recursive call will randomly choose the worst possible pivot.

A sorted and a reverse-sorted input are no longer worst-case scenarios: both are sorted in log-linear time due to the random pivot choice.

14.7.2 Quickselect

Next I'm going to show a decrease-and-conquer adaption of quicksort to solve a different problem.

Consider the **selection problem**: find the k -th smallest item in a non-empty unsorted sequence, with $0 < k \leq n$. For example, if $k = 1$ then we're looking for the minimum and if $k = n$ then we're looking for the maximum.

If we know that there will be many queries on the same sequence, then it's best to sort it once and return the k -th item for each query. Let's assume we don't know that and thus must solve the selection problem without sorting.

The **quickselect** algorithm adapts two-way quicksort. It only recursively searches the partition that includes the sought item, discarding the other partition. How does it know where the item is?

Well, if partition *smaller* has $k - 1$ items, then the pivot, which is the next larger item, is the k -th smallest item. This is a base case: the algorithm returns the pivot without recurring into either partition.

If partition *smaller* has k or more items, then the k -th smallest must be there, so the algorithm recurs into it and ignores partition *larger*.

Finally, if partition *smaller* has fewer than $k - 1$ items, the sought item is in the other partition. But it's not the k -th smallest item of that partition. Let's suppose we're looking for the 17th smallest item among 20 items and that partition *smaller* has 14 items. Together with the pivot, we can discard 15 items. The sought item is thus the second smallest in partition *larger*. More generally, if *smaller* has s items, we search for the $k-s-1$ -th smallest item in *larger*.

It has been proven that on average quickselect has linear complexity.

Exercise 14.7.2

What kind of decrease and conquer is quickselect?

Hint Answer

Exercise 14.7.3

Here again is the quicksort algorithm. The *pivot* returned by the auxiliary function is a single-item sequence.

1. if $n < 2$:
 1. let *sorted* be *unsorted*
2. otherwise:
 1. let $(\text{smaller}, \text{pivot}, \text{larger})$ be *partition(unsorted)*
 2. let *sorted* be *quicksort(smaller, key)* concatenated with *pivot* and *quicksort(larger, key)*

Modify the above to become the quickselect algorithm. You can assume the function call is *quickselect(unsorted, key, k)* with a non-empty *unsorted* sequence and $0 < k \leq n$.

Hint Answer

14.8 Pigeonhole sort

The algorithms presented so far are **comparison sorts**: they're based on comparing pairs of items, or their keys. This section shows an example of a **distribution sort**, which first

distributes the items based on their keys and then collects them in order. The reason for a different kind of sorting algorithm is efficiency.

14.8.1 Comparison sort complexity

Earlier we assumed that sorting is linear in the best case, to check the input is already sorted, and quadratic in the worst case: comparing all items to each other should be enough to put each item in its correct place in the sequence.

Insertion sort is linear in the best case and quadratic in the worst case, as it has to compare each unsorted item to at least one and at most all previously sorted items. Selection sort is always quadratic because it compares each unsorted item to all other unsorted items.

Our earlier argument only posited that sorting doesn't need more than quadratic time, but as we have meanwhile seen, human imagination is able to come up with worse solutions than needed (like bogosort) and with better solutions than expected (like merge sort, which is always log-linear). It turns out that log-linear is the lowest worst-case complexity for comparison sorts, as I'll show next.

Let's assume the n items have unique keys. Therefore, there's a single ascending permutation. As bogosort did, we can see sorting as searching for the one permutation that is sorted.

If for example we find out that $\text{key}(A) < \text{key}(B)$, we can discard all permutations where item A comes after (to the right of) item B, because those aren't sorted. For each permutation to be removed from the search space, because A and B are in the wrong order, there's exactly one other permutation that is kept in the search space, because it's equal except that A and B swapped places and hence are in the correct order. In summary, every comparison gives enough information to discard half the search space.

After one comparison the search space has $n! / 2$ permutations. After two comparisons it has $n!/2^2 = n!/2^2$ permutations. In general, after c comparisons, the search space has size $n!/2^c$. The algorithm stops when the search space has one permutation (the sorted one), i.e. when $n!/2^c = 1$, which is the same as $n! = 2^c$ or $c = \log n!$. It has been proven that $\log n!$ is about the same as $n \log n$ and so that's the number of comparisons needed.

14.8.2 Algorithm

If we assume more about the sorting keys than just being comparable, then we can use more efficient sorting algorithms, like **pigeonhole sort**.

The algorithm gets its name from the pigeonholes used in mail sorting. You can see them at start of the second part of the [BBC programme](#). All mail to the same postcode, the sorting key, goes in the same pigeonhole. We need to know in advance the possible keys to create a pigeonhole for each.

Pigeonhole sort uses a map of keys to items with that key. This can be done with a lookup table of length k and a key function that returns natural numbers from 0 to $k - 1$. In the first phase (step 3 below) the algorithm distributes the items according to their keys. In the second phase (steps 4 and 5) it collects the items from lowest to highest key.

1. let *pigeonholes* be ()
2. repeat k times:

1. append an empty collection to *pigeonholes*
3. for each *item* in *unsorted*:
 1. add *item* to *pigeonholes*[*key(item)*]
4. let *sorted* be ()
5. for each *slot* in *pigeonholes*:
 1. for each *item* in *slot*:
 1. append *item* to *sorted*

14.8.3 Complexity

The algorithm only uses constant-time operations. Step 2.1 is done k times, steps 3 and 5.1.1 are each executed n times to add all items in unsorted order and retrieve them in sorted order. The complexity is thus $\Theta(n + k)$. Usually k is either a constant or a multiple of n , so the complexity of pigeonhole sort is linear in the length of the input.

Pigeonhole sort is based on the same idea as selection sort: at each step append the smallest unsorted item to the sorted items. The difference is that it uses natural number keys and a lookup table to select the minimum in constant instead of linear time. This enables pigeonhole sort to process n items in linear time, while selection sort takes quadratic time.

14.8.4 Code and tests

Pigeonhole requires an extra parameter to know how many slots to create.

```
[1]: from typing import Callable

def pigeonhole_sorted(unsorted: list, key: Callable, slots: int) -> list:
    """Return a permutation with keys in non-decreasing order.

    Precondition: for each item in unsorted 0 <= key(item) < slots
    """
    pigeonholes = []
    for slot in range(slots):
        # noqa: B007
        pigeonholes.append([])
    for item in unsorted:
        pigeonholes[key(item)].append(item)
    result = []
    for slot in pigeonholes:
        for item in slot:
            result.append(item)
    return result
```

I can't reuse the *same tests* as for the previous sorting algorithms because of the extra *slot* parameter and because the key functions don't return natural numbers from 0 onwards. I must

redefine the key functions and the test table. I will however use the same problem instances.

```
[2]: from algoesup import test

%run -i ../m269_sorting

def value_nat(card: str) -> int:
    """Return 0 to 12 for value A2...9TJQK respectively.

    Preconditions: as for function 'suit'
    """
    return value(card) - 1 # the value function returns 1 to 13

def suit_nat(card: str) -> int:
    """Return 0 to 3 for suit 'C', 'D', 'H' and 'S' respectively.

    Preconditions: as for function 'suit'
    """
    return {"C": 0, "D": 1, "H": 2, "S": 3}[card[1]]

def card_nat(card: str) -> int:
    """Return 0 to 51 according to the sorted order of the card.

    Cards are sorted first by suit, then by value.
    Preconditions: as for function 'suit'
    """
    return suit_nat(card) * 13 + value_nat(card)

pigeonhole_sorted_tests = [
    # case,           unsorted,          key,      slots,   sorted
    ('no cards',     [],                card_nat, 52, []),
    ('1 card',       ['AS'],            card_nat, 52, ['AS']),
    ('same card',    ['6D', '6D'],      card_nat, 52, ['6D', '6D']),
    ('3 cards',      ['JC', '8D', 'TS'], value_nat, 13, ['8D', 'TS', 'JC'
    ↵']),
    ('value up',     UP_DOWN,          value_nat, 13, UP_DOWN),
    ('suit down',    UP_DOWN,          suit_nat, 4, ['KC', 'QD', '3H',
    ↵'AS']),
    ('same value',   SAME_VALUE,       card_nat, 52, ['TC', 'TD', 'TH',
    ↵'TS']),
]
test(pigeonhole_sorted, pigeonhole_sorted_tests)
```

```
Testing pigeonhole_sorted...
Tests finished: 7 passed (100%), 0 failed.
```

14.8.5 Performance

Pigeonhole sort takes linear time on all inputs. Insertion sort is also linear when the input is sorted. Which do you expect to be faster for sorted inputs: insertion sort or pigeonhole sort?

Insertion sort uses no extra data structure. It does no swaps and only one comparison if the item is in its sorted place, so it should be faster. Let's eat the pudding, so to speak.

```
[3]: for doubling in range(5):
    items = list(range(100 * 2**doubling))
    %timeit -r 5 pigeonhole_sorted(items, identity, len(items))

10.8 µs ± 11.2 ns per loop (mean ± std. dev. of 5 runs, 100,000 loops
˓→each)
21.8 µs ± 39.2 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
˓→each)
44.2 µs ± 83.1 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
˓→each)
92.6 µs ± 293 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
˓→each)
187 µs ± 333 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
˓→each)
```

Comparing these values to *those for insertion sort*, the latter is indeed slightly faster.

Pigeonhole not only has better complexity than comparison sorts, it's also simpler: there are no recursive calls, swaps or partitions. It's thus quite faster than our versions of *merge sort* or *quicksort*, which aren't in-place either.

14.9 Summary

For the purposes of M269, sorting consists of putting a sequence of items in non-decreasing order of their keys. Python's built-in `sort` method and `sorted` function can take a parameter named `key` indicating the function that returns an item's key.

In-place sorting modifies the input sequence and only uses a constant amount of additional memory.

A **comparison sort** algorithm compares the keys of two items at a time, whereas a **distribution sort** algorithm first separates the unsorted items according to their keys and then collects them from lowest to highest key, without ever comparing keys. All algorithms below are comparison sorts, except for pigeonhole sort.

14.9.1 Algorithms

Bogosort is an exhaustive search algorithm that generates each permutation of the input sequence and tests whether it's sorted.

Insertion sort and **selection sort** both move one item at a time from an unsorted part to a sorted part. Insertion sort takes the next unsorted item and finds its place in the sorted part. Selection sort finds the smallest unsorted item and puts it at the end of the sorted part, or finds the largest unsorted item and puts it at the start of the sorted part.

Merge sort and **quicksort** are recursive divide-and-conquer algorithms. Merge sort partitions the input sequence into two halves, recursively sorts them and merges them. Quicksort chooses a pivot item and partitions the other items into those smaller than the pivot and those larger than or equal to the pivot. The recursively sorted partitions are concatenated, with the pivot in between. A three-way variant of quicksort creates a third partition that has all items with the same key as the pivot.

Quickselect is a modified quicksort algorithm that solves the **selection problem**: find the k -th smallest item in an unsorted sequence. Quickselect is a recursive decrease-and-conquer algorithm that searches only one of the partitions, depending on their size. Its average complexity is linear.

Pigeonhole sort is a distribution sort that creates a lookup table with enough slots for all the keys. Each item is appended to the slot corresponding to its key. Finally, items are collected from the first to the last slot. The algorithm requires keys to be natural numbers within a bounded range.

14.9.2 Complexities

The following table shows the characteristics of the implementations presented in the previous sections, where n is the length of the input sequence and k is the number of different keys. The last row shows the characteristics of Python's built-in algorithm. The complexities listed assume that it takes constant time to obtain the key of an item and compare two keys.

Algorithm	In-place	Best case	Worst case
Bogosort	no	$\Theta(n)$	$\Theta(n! \times n)$
Insertion sort	yes	$\Theta(n)$	$\Theta(n^2)$
Selection sort	yes	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	no	$\Theta(n \log n)$	$\Theta(n \log n)$
Two-way quicksort	no	$\Theta(n \log n)$	$\Theta(n^2)$
Three-way quicksort	no	$\Theta(n)$	$\Theta(n^2)$
Pigeonhole sort	no	$\Theta(n + k)$	$\Theta(n + k)$
Powersort	yes	$\Theta(n)$	$\Theta(n \log n)$

Quicksort has quadratic complexity when the smallest or largest item is chosen at each step. To avoid this, choose a random pivot or the median of three items.

Quicksort has log-linear complexity in the average case. This can be observed with a sequence of items in random order, e.g. as produced by function `shuffle` in module `random`.

An in-place version of quicksort is faster but has the same complexities.

Comparison sorts can't take less than linear time in the best case and less than log-linear time in the worst case, because each comparison reduces the search space (initially, all permutations of the input) by half at most.

CHAPTER 15

TMA 02 PART 1

This study-free week is for you to catch up if you need to, and to complete the first part of TMA 02. If you haven't done so yet, download TMA 02 from the 'Assessment' tab of the M269 website, and put it in a `TMA02` subfolder of your M269 folder.

Before attempting TMA 02, I suggest you look again at your tutor's feedback on TMA 01.

This chapter complements the summaries of Chapters [11 \(exhaustive search\)](#), [12 \(recursion\)](#), [13 \(divide and conquer\)](#) and [14 \(sorting\)](#) with higher level advice for designing and analysing algorithms.

The guidance in [Chapter 5](#) and [Chapter 10](#) still applies.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

15.1 Brute-force search

To apply brute-force search, you should ask yourself:

- What type of problem is it?
- What are the candidates?
- How can I generate and test them?
- What should I do with each solution found?

15.1.1 Problem type

If you can rephrase the problem in one of the following forms, you can solve it with brute-force search; otherwise, you may have to look for a different approach.

1. Search problem: Find all/one/at most n ... that ..., e.g. find all products in store that are size 7 shoes.

2. Optimisation or constraint satisfaction problem (CSP): Find all/one/at most n of the smallest/largest/cheapest/etc. ... that ..., e.g. find any of the cheapest products in store that are size 7 shoes.
3. Decision problem: The input has/is ... if and only if a ... that ... can/cannot be found, e.g. the item is the cheapest if and only if a product that has a lower price cannot be found.
4. Counting problem: How many ... are ...?, e.g. how many products in the store are size 7 shoes?

15.1.2 Candidates

If you can apply brute-force search, the next step is to think what the candidates are.

1. Are the candidates one of the inputs?
2. Are the candidates integers? If so, what is the range of candidates, i.e. what are the smallest and largest integers?
3. Are the candidates items, indices or slices of a sequence?
4. Are the candidates tuples or sets of items? This is often the case in CSPs, which ask for multiple items that together satisfy the constraints.
5. Do the candidates include the solutions sought? Otherwise, the algorithm will produce the wrong output.
6. Is there a finite number of candidates? Otherwise, the algorithm can't generate all of them.

If you rephrased the problem as ‘find all C that ...’, or something similar, then the candidates are C or some of them. For example, the factorisation problem can be stated as ‘find all positive integers that divide the given number’, but there are infinite positive integers, so we must restrict the candidates to be only some of the positive integers.

15.1.3 Generate

Once you know the candidates, think how to generate them. The fewer candidates the algorithm generates, the faster it is. Pruning the search space has the greatest effect on the algorithm’s run-time.

1. If the candidates are items in a sequence or integers in a range, use a for-loop.
2. If the candidates are tuples, use nested loops or iterate over permutations or subsets.
3. Is it possible to stop early, i.e. is there a point where no further solutions can be found? This may be the case if candidates are sorted.
4. Is it possible to avoid generating some solutions by computing them from others?
5. Is it possible to avoid generating candidates that will be rejected anyhow?

Examples of problems to which the above apply are:

1. find size 7 shoes in a given store; find factors from 1 to n
2. find pairs of products to spend a voucher; find the shortest tour; find the most valuable subset of items that fit in a knapsack

3. stop after size 7 shoes when going through a sorted store
4. generate factors from 1 to \sqrt{n} and compute the others
5. for the voucher problem, don't generate the symmetric product pair.

15.1.4 Test

The next step is to think about how to test each candidate.

- What are a candidate's properties for it to be a solution?
- Can the candidate be tested in isolation, or must it be compared to other candidates? For optimisation problems, test if the candidate is better than the solutions found so far.
- Is the test complicated? If so, consider it as a separate decision problem, solved with an auxiliary function.
- Can the test be done in constant time?
- Can the test be simplified by generating candidates that already satisfy some criterion? For example, by generating non-symmetric product pairs, the test just checks if their prices add up to the voucher.

If you rephrased the problem as 'find all C that T ', or something similar, then T is the test, i.e. the criterion that candidates C must satisfy to be solutions.

15.1.5 Solutions

The final step is to think of what to do with each solution found.

- Does the problem ask for all, any one, or some of the solutions? If it doesn't ask for all solutions, then the algorithm can stop when enough solutions are found, provided it's not an optimisation problem. If it is, it can't stop as there may be better solutions ahead.
- Does the problem ask for the number of solutions but not the solutions themselves? If so, the algorithm may not need to keep a collection of solutions.
- Is it an optimisation problem? If so, the algorithm must check if the new solution found is worse, as good, or better than the solutions so far and process it accordingly.

15.1.6 Algorithm

The exhaustive search algorithm depends on the answers to the above questions, but it's usually some variation on the basic generate-and-test template:

1. let $solutions$ be the empty sequence
2. for each $candidate$ in $candidates$:
 1. if $candidate$ is a solution:
 1. add $candidate$ to $solutions$

For example, if the problem asks only for the number of solutions, then the sequence of solutions is replaced by an integer counter that starts at zero. If the problem asks for the first

10 solutions, then the algorithm stops if the sequence reaches that length. If the problem asks for unique solutions, then they should be kept in a set instead of a sequence. If it's a CSP, then step 2 probably iterates over all permutations or subsets, or has nested loops. And so on. Patterns of linear search that can be further adapted are in Sections 5.2 and 11.1.

15.1.7 Complexity

The complexity of exhaustive search is usually the number of generated candidates multiplied by the complexity of generating and testing each one.

In the worst case, the number of candidates is usually n , $n!$ or 2^n , where n is the size of the input collection. This depends on whether the candidates are respectively the input items themselves, or permutations or subsets thereof.

Testing each candidate usually takes constant time or linear time in the size of the candidate. If the size of each candidate is bounded, e.g. if each candidate is a word in some language, then testing takes constant rather than linear time.

Sorting can be done in log-linear time. Take that into account to decide whether it's worth sorting the input before searching.

15.1.8 Performance

When measuring run-times, use the following table to check the algorithm has the expected complexity.

If the run-time ...	when the input size ...	then the complexity is ...
stays the same	grows	constant
grows by a fixed amount	doubles	logarithmic
doubles	doubles	linear
quadruples	doubles	quadratic
multiples by 8	doubles	cubic
grows by a fixed factor	grows by 1	exponential
grows by increasing factors	grows by 1	factorial

15.2 Divide and conquer

A recursive divide-and-conquer algorithm follows these steps:

1. if the input is a base case, compute the output directly and stop
2. divide the input into $p > 1$ parts
3. recursively apply the algorithm to conquer, i.e. solve, each part
4. combine the subsolutions to get the solution for the whole input.

If the algorithm divides the input in two parts, one of which has a fixed size, typically one, and recursively processes the other, larger, part in step 3, then it's a decrease-by-constant-amount

algorithm. Decrease-by-constant-amount algorithms should be iterative whenever possible as recursive ones run the risk of exceeding the call stack capacity.

If the algorithm divides the input into equally sized parts but only processes one of them in step 3, then it's a decrease-by-constant-factor algorithm, where the factor is p .

The key questions to see if a problem can be solved by a divide- (or decrease-) and-conquer algorithm are:

- Are there any inputs small enough to be solved directly?
- Is there an easy and efficient way to partition the input?
- Can subsolutions be easily combined?

The answers to these questions lead to steps 1, 2 and 4 above. For example, insertion and merge sort make partitioning easy, while selection sort and quicksort make combining easy.

If the input data type can be defined recursively, like the head and tail of a sequence, then a divide- or decrease-and-conquer algorithm is often the natural choice, because it can follow the recursive structure of the data.

If the input data is an ordered sequence, e.g. a range of numbers, and the problem is a search problem, consider using some form of binary search. If the input isn't sorted, you may still use a decrease-and-conquer algorithm if the properties of the partitions or of the chosen pivot allow you to easily decide which partition to search. Quickselect is an example.

15.2.1 Complexity

For a recursive decrease-and-conquer algorithm that decreases the input of size n by one, define the complexity as follows:

- if $n \leq s$: $T(n) = \Theta(b)$
- if $n > s$: $T(n) = \Theta(d) + T(n - 1) + \Theta(c)$

where

- s is the size of the largest base case, usually 0 or 1
- $\Theta(b)$ is the complexity of handling the base cases, usually $\Theta(1)$ because their size is bounded
- $\Theta(d)$ is the complexity of decreasing the input
- $\Theta(c)$ is the complexity of computing the solution from the subsolution for $n - 1$
- d and c are either 1 or expressions in n .

For the recursive factorial algorithm $d = c = 1$, but for recursively computing the length of a sequence with slicing $d = n$ and $c = 1$.

If a recursive algorithm divides the input into $p > 1$ partitions of equal size (the best case) and processes one or more of them, then the complexity definition is of the form

- if $n \leq s$: $T(n) = \Theta(b)$
- if $n > s$: $T(n) = \Theta(d) + r \times T(n / p) + \Theta(c)$

where

- p is usually 2 or 3
- r is the number of recursive calls, with $1 \leq r \leq p$
- $\Theta(d)$ is the complexity of creating the partitions
- $\Theta(c)$ is the complexity of combining the subsolutions.

Usually $r = 1$ for a decrease-and-conquer algorithm that decreases the input by a constant factor, like binary search, and $r = p$ for a divide-and-conquer algorithm, like two-way quicksort.

Usually $d = n$ when using slicing and $d = 1$ when passing the range of indices.

Once you write the recursive definition, you can look up the complexity in the following table, assuming the base cases take constant time.

If $T(n) = \dots$	then the complexity is ...	Example
$T(n - 1) + \Theta(1)$	$\Theta(n)$	factorial
$T(n - 1) + \Theta(n)$	$\Theta(n^2)$	length of sequence with slicing
$T(n/p) + \Theta(1)$	$\Theta(\log n)$	binary search without slicing
$T(n/p) + \Theta(n)$	$\Theta(n)$	binary search with slicing
$p \times T(n/p) + \Theta(1)$	$\Theta(n)$	maximum without slicing
$p \times T(n/p) + \Theta(n)$	$\Theta(n \log n)$	merge sort

CHAPTER 16

ROOTED TREES

So far we organised items in some kind of sequence, like stacks and queues, or as an unordered ‘bunch’, like maps and sets. All collections were ‘flat’.

A **rooted tree** is an ADT that can represent a hierarchical collection, in which items are organised into levels. In a rooted tree, only one item (called the **root**) can be in the top level. For example, folders on a disk are organised hierarchically in a tree. Each folder contains zero or more subfolders, which in turn may contain others. The root folder on a disk is usually named / or \.

This chapter introduces rooted trees and their properties, and how to represent and manipulate them, including two new forms of search in order to find items in a tree: breadth- and depth-first search. You will learn about two kinds of rooted trees: binary search trees and heaps. The latter provide an efficient implementation of priority queues, which in turn leads to heapsort, a sorting algorithm with log-linear worst-case complexity.

To understand and handle rooted trees, you will draw on concepts and techniques introduced before: recursion, divide and conquer, stacks, queues, linked lists, binary search and logarithmic complexity.

Trees without a designated root item are introduced in the next chapter.

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn about rooted trees and breadth- and depth-first search.
- Develop and apply algorithms and data structures to solve computational problems - the new data structures and algorithms require applying previous ones.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

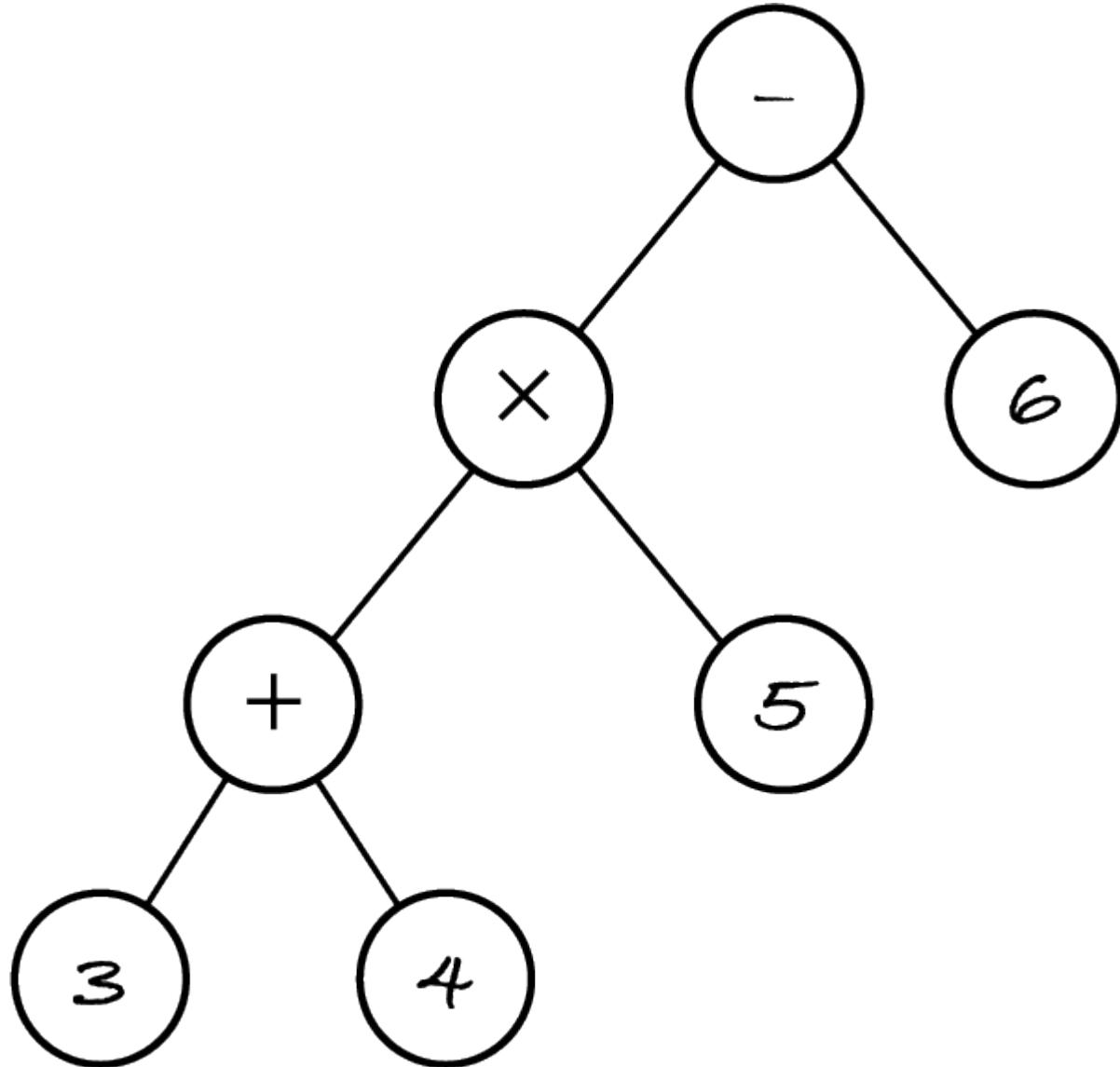
16.1 Binary tree

In M269 we focus on a restricted form of rooted trees: binary trees. They can be defined recursively:

A **binary tree** is either empty or it consists of an item, called the **root**, and two binary trees, called the **left subtree** and the **right subtree**.

The next figure shows a binary tree that represents the expression $(3+4) \times 5 - 6$. Rooted trees are usually depicted from the root downwards, unlike natural trees.

Figure 16.1.1



The root is the subtraction operator. The left subtree's root is the multiplication operator. The right subtree consists of root 6 and two empty subtrees. Why the expression is represented in this particular way will become clear in the next section, when we evaluate such expression trees.

Many examples, like the folder hierarchy on a disk, can't be modelled as binary trees, because a folder can have more than two subfolders. But the concepts and techniques for binary trees can be extended to rooted trees with any number of subtrees.

16.1.1 Terminology

A **node** consists of an item and the references to the left and right subtrees. The **size** of a tree is the number of its items (or nodes); the example tree has size 7.

A node A is the **parent** of node B, and B is the **left or right child** of A, if B is the root of the left or right subtree of A, respectively. For example, node 6 is the right child of the subtraction node, which is also the parent of the multiplication node.

Every node has exactly one parent, except the root, which has no parent. A **leaf** is a childless node, i.e. both its subtrees are empty. In the example, the leaves are the integer literals.

A node A is an **ancestor** of node B, and B is a **descendant** of A, if A is a parent of B or A is an ancestor of the parent of B. For example, the ancestors of node 3 are addition, multiplication and subtraction. A subtree rooted at a node A consists of A and all its descendants. For example, the subtree rooted at the addition node consists of it and the descendants 3 and 4.

The **level** or **depth** of a node is the number of its ancestors. The root has depth 0 because it has no ancestors. In the example tree, node 5 and the addition node have depth 2; nodes 3 and 4 are at level 3.

The **height** of a tree is the number of levels. The example tree has four levels, so its height is 4. An empty tree has no levels, so its height is 0.

The height of a non-empty tree is one more than the largest depth of all nodes. The largest depth in the example tree is 3 (for the left-most leaves), so the height is 4. A tree with only one node (the root, at depth 0) has height 1.



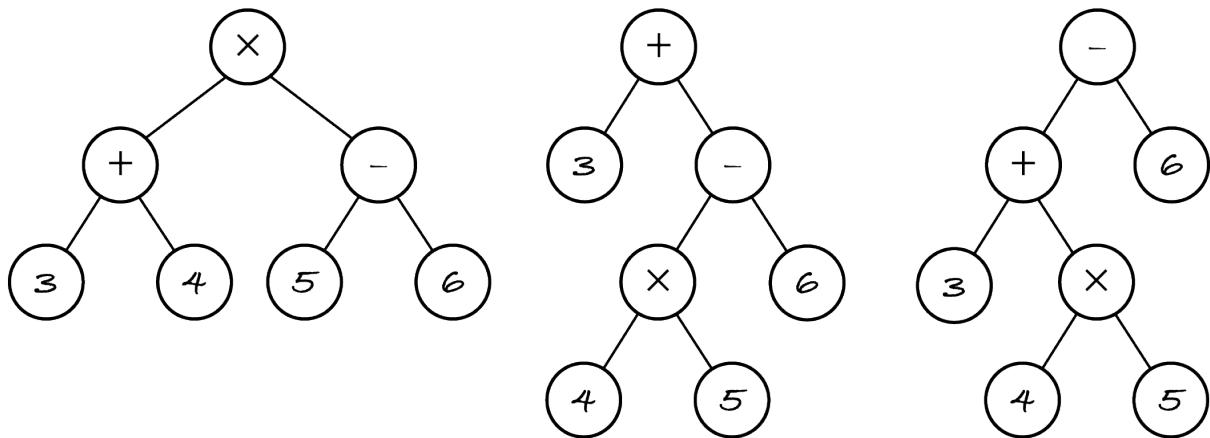
Info: Many authors define the height of the empty tree as -1 and the height of a non-empty tree as the largest depth.

A binary tree is **perfect** if all its levels are full, i.e. all parents have two children and all leaves are at the same depth. An empty tree and a tree with just one node are perfect trees. The left-hand tree in the next figure is also perfect.

Exercise 16.1.1

Consider the following binary trees, which represent, from left to right, expressions $(3+4)\times(5-6)$, $3+((4\times 5)-6)$ and $(3+(4\times 5))-6$.

Figure 16.1.2



For each tree:

1. State its size and height.
2. List the nodes at level 2.
3. Explain if the multiplication node is a descendant of the subtraction node.

Answer

16.1.2 ADT and data structure

The binary tree ADT has the following operations:

Operation	Effect	Algorithm in English
new	create a new empty binary tree	let t be an empty binary tree
join	create a tree from item i and trees l and r	join(i, l, r)
root	obtain the root item of non-empty tree t	root(t)
left	obtain the left subtree of non-empty tree t	left(t)
right	obtain the right subtree of non-empty tree t	right(t)
is empty	check if a given tree is empty	t is empty

The join operation puts together a tree while the root, left and right operations take it apart. We have $\text{root}(\text{join}(i, l, r)) = i$, $\text{left}(\text{join}(i, l, r)) = l$ and $\text{right}(\text{join}(i, l, r)) = r$. This is the same approach as for composing sequences with the prepend operation and decomposing them with the head and tail operations.



Note: When designing a data type made of several parts, include operations to join and to separate the parts.

Instead of writing in one go a large class with many methods, I define a data structure and add operations one by one, as standalone functions. Introducing operations incrementally makes it easier for me to explain (and for you to learn) how binary trees work.

The binary tree data structure follows the recursive definition: a tree node has an item and points to two children nodes. A binary tree is like a bifurcating *linked list*. I represent an empty tree by a node without a root or subtrees.

[1]: # this code is also in m269_tree.py

```
class Tree:
    """A rooted binary tree."""

    def __init__(self) -> None:
        """Create an empty tree."""
        self.root = None
        self.left = None
        self.right = None
```

The class could be named `BinaryTree` or `TreeNode` but since we're using only one kind of tree, I prefer a shorter name.

Let's implement the ADT operations. The new operation is provided by the constructor. To obtain the root, left and right subtrees we can simply access the corresponding attribute because we're using the class as the raw data structure. The remaining operations are:

[2]: # this code is also in m269_tree.py

```
def is_empty(tree: Tree) -> bool:
    """Return True if and only if tree is empty."""
    return tree.root == tree.left == tree.right == None

def join(item: object, left: Tree, right: Tree) -> Tree:
    """Return a tree with the given root and subtrees."""
    tree = Tree()
    tree.root = item
    tree.left = left
    tree.right = right
    return tree
```

We construct trees bottom-up, starting from the leaves and joining two subtrees. Here's how to construct the 3+4 subtree of Figure 16.1.1.

[3]: THREE = join(3, Tree(), Tree()) # a leaf has empty subtrees
 FOUR = join(4, Tree(), Tree())
 SUM = join("+", THREE, FOUR) # the subtree for 3 + 4

To create trees more easily, let's define a convenience operation.

```
[4]: # this code is also in m269_tree.py

def leaf(item: object) -> Tree:
    """Return a node with the item and empty subtrees."""
    return join(item, Tree(), Tree())

# The leaves for all the example trees.
THREE = leaf(3)
FOUR = leaf(4)
FIVE = leaf(5)
SIX = leaf(6)
```

Now we can create the three trees in Figure 16.1.2, named after the order in which the plus, minus and times operators appear in the `join` arguments.

```
[5]: # this code is also in m269_tree.py

TPM = join("*", join("+", THREE, FOUR), join("-", FIVE, SIX)) #_
↪ (3+4)*(5-6)
PMT = join("+", THREE, join("-", join("*", FOUR, FIVE), SIX)) #_
↪ 3+((4*5)-6)
MPT = join("-", join("+", THREE, join("*", FOUR, FIVE)), SIX) #_
↪ (3+(4*5))-6
```

I can reuse the same leaf objects for different trees because they won't be modified (hence the uppercase names).

Exercise 16.1.2

Write the Python expression for the tree in Figure 16.1.1, for expression $((3+4)\times 5) - 6$.

```
[6]: %run -i ../m269_tree
```

```
pass
```

Answer

Before we move on, here's one more operation to illustrate accessing the subtrees.

```
[7]: # this code is also in m269_tree.py
```

```
def is_leaf(tree: Tree) -> bool:
    """Return True if and only if the tree is a single leaf."""
    return not is_empty(tree) and is_empty(tree.left) and is_
↪ empty(tree.right)
```

```
[8]: is_leaf(THREE)
```

```
[8]: True
```

```
[9]: is_leaf(Tree())
```

```
[9]: False
```

```
[10]: is_leaf(TPM)
```

```
[10]: False
```

16.2 Algorithms on trees

Algorithms on binary trees usually follow a *divide-and-conquer approach* to process both subtrees and thereby all nodes. This takes linear time in the size of the tree, assuming that processing each node takes constant time. Let's see a concrete example: computing the size of a tree.

16.2.1 Divide and conquer

Due to the recursive definition of binary trees, a function f on them is usually defined recursively like this:

1. if $tree$ is empty: $f(tree) = \dots$
2. otherwise: $f(tree) = \text{an expression based on operations root, left, right and join.}$

To come up with such a definition you need to answer these questions:

1. What's the output for an empty tree?
2. If I know the outputs for the left and right subtrees, what's the output for the whole tree?

For example, the size of the empty tree is zero, and if I know the sizes of the left and right subtrees, then the size of the tree is their sum plus one, for the root.

- if $tree$ is empty: $\text{size}(tree) = 0$
- otherwise: $\text{size}(tree) = \text{size}(\text{left}(tree)) + \text{size}(\text{right}(tree)) + 1$

The recursive definition of the length of a sequence didn't refer to the head of the sequence; similarly, the size of a tree doesn't refer to the root.

Like for sequences, recursive definitions on trees are straightforward to translate to code. First we must 'import' the definition of `Tree`.

```
[1]: %run -i ../m269_tree
```

Now we can define a new operation on binary trees.

```
[2]: # this code is also in m269_tree.py
```

```
def size(tree: Tree) -> int:
    """Return the number of nodes in tree."""
    if is_empty(tree):
        return 0
    else:
        return size(tree.left) + size(tree.right) + 1
```

I test the function on one expression tree, as they all have the same size.

```
[3]: size(TPM)
```

```
[3]: 7
```

Exercise 16.2.1

Recursively define the height of a tree.

- if *tree* is empty: $\text{height}(\text{tree}) = \dots$
- otherwise: $\text{height}(\text{tree}) = \dots$

Hint Answer

Exercise 16.2.2

Implement the operation.

```
[4]: %run -i ../m269_tree
from algoesup import test
```

```
def height(tree: Tree) -> int:
    """Return the height of the tree."""
    pass

height_tests = [
    # case,           tree,     height
    ('empty tree',   Tree(),   0),
    ('(3+4)*(5-6)', TPM,     3),
    ('3+((4*5)-6)', PMT,     4),
    ('(3+(4*5))-6', MPT,     4),
]
test(height, height_tests)
```

Answer

16.2.2 Arm's-length recursion

The size algorithm always does two recursive calls per node, whether a node has 0, 1 or 2 children. However, empty subtrees don't add anything to the size of the tree. Making a recursive call to immediately return zero seems a bit pointless.

Arm's-length recursion checks for the base case *before* making a recursive call. For the size function, this means checking if a subtree is empty and not making a recursive call if it is. Since one or both subtrees may be empty, we must check three additional cases. The base case must still be checked in case the whole tree is empty.

```
[5]: def size_arm(tree: Tree) -> int:
    """Return the size of the tree using arm's length recursion."""
    if is_empty(tree):
        return 0
    elif is_leaf(tree): # both subtrees empty
        return 1
    elif is_empty(tree.left): # left subtree empty
        return size_arm(tree.right) + 1
    elif is_empty(tree.right): # right subtree empty
        return size_arm(tree.left) + 1
    else:
        return size_arm(tree.left) + size_arm(tree.right) + 1
```

The new algorithm is longer, inelegant, repetitive and thus prone to typos and other errors. It only recurs on non-empty subtrees, so it makes as many recursive calls as there are nodes, not twice as much, but each call makes more checks. Let's compare this version to the first one, using a tall tree with one child per node; essentially, a linked list.

```
[6]: tree = leaf("last node")
for level in range(1000): # noqa: B007
    tree = join("a parent node", tree, Tree())

%timeit -r 5 size(tree)
%timeit -r 5 size_arm(tree)

250 µs ± 664 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops→each)
458 µs ± 1.58 µs per loop (mean ± std. dev. of 5 runs, 1,000 loops→each)
```

In this example, arm's length recursion takes longer, even though it makes fewer recursive calls.



Note: Avoid arm's length recursion: it complicates your code and usually slows it down.

If an operation isn't defined for the empty tree, then an algorithm must first check if a subtree isn't empty before making a recursive call. Consider finding the largest item in a binary tree.

The preconditions are that the input tree isn't empty and its items are comparable.

- if $tree$ is a leaf: $\text{largest}(tree) = \text{root}(tree)$
- if $\text{left}(tree)$ is empty and $\text{right}(tree)$ isn't: $\text{largest}(tree) = \max(\text{largest}(\text{right}(tree)), \text{root}(tree))$
- if $\text{right}(tree)$ is empty and $\text{left}(tree)$ isn't: $\text{largest}(tree) = \max(\text{largest}(\text{left}(tree)), \text{root}(tree))$
- otherwise: $\text{largest}(tree) = \max(\text{largest}(\text{left}(tree)), \text{largest}(\text{right}(tree)), \text{root}(tree))$

This is *not* arm's-length recursion: each recurrence relation is checking for the empty tree, not for the base case (tree is a leaf). The definition is making sure no recursive call violates the preconditions.

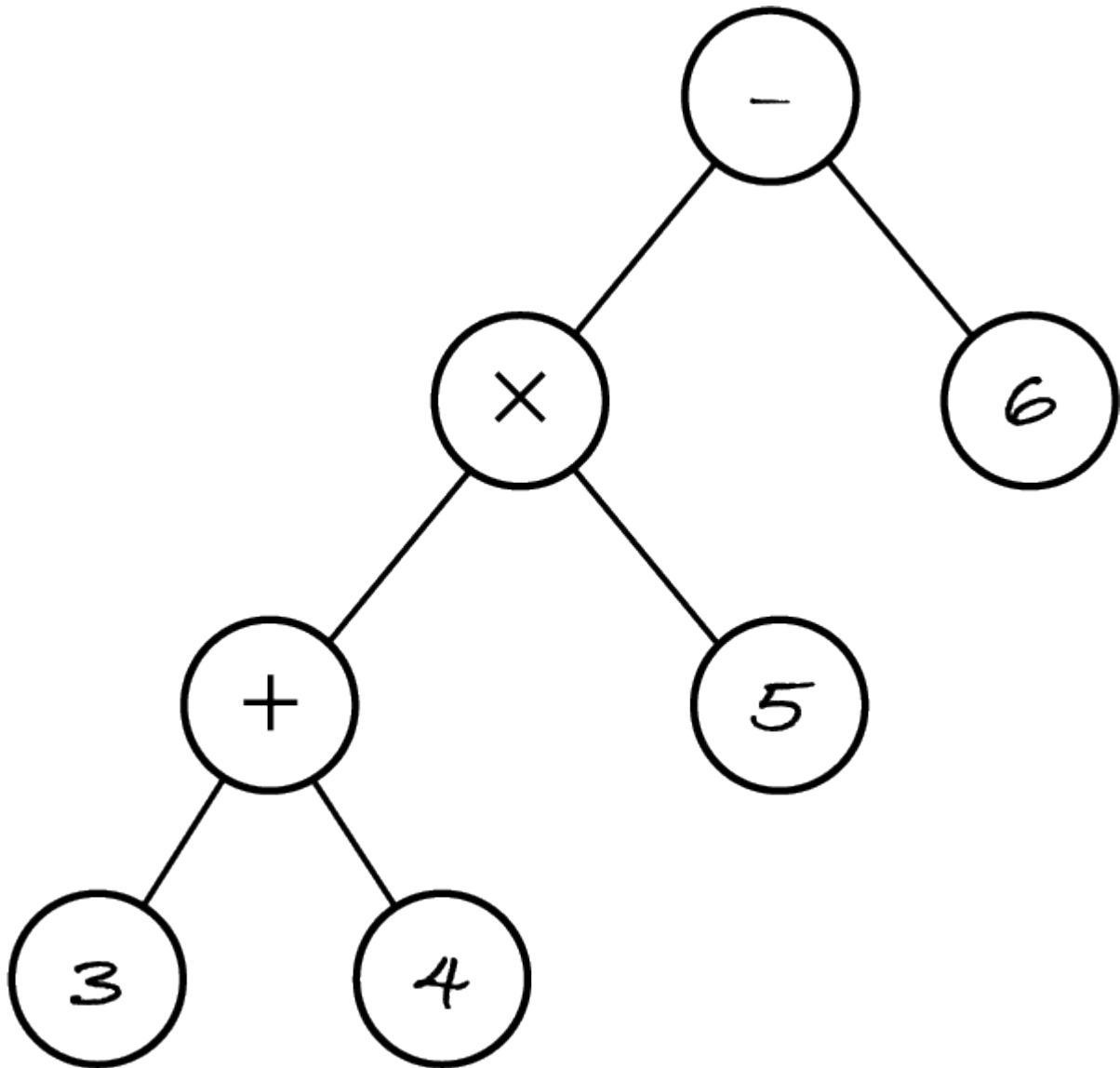
16.3 Traversals

To do a brute-force search over a binary tree we must traverse it to visit all its nodes. The traversal generates the candidates and visiting a node does the testing, i.e. checks if it satisfies the search criteria.

Traversals can be used for non-search problems too, e.g. to print the tree. Visiting a node processes its item and what is done depends on the problem, e.g. the size operation process a node by adding one to count.

There are several systematic ways of traversing a binary tree. Once again I'll use the tree for expression $(3+4)\times 5 - 6$ to illustrate the concepts.

Figure 16.3.1



[1]: %run -i ./m269_tree

```
# MTP is short for minus, times, plus
MTP = join("-", join("*", join("+", THREE, FOUR), FIVE), SIX)
```

16.3.1 Depth-first search

Traversals can be classified according to whether the root is visited before, in between or after visiting the subtrees.

A **pre-order** traversal first visits the root, then the left subtree, then the right subtree. For the example tree, it would visit nodes $-$, \times , $+$, 3 , 4 , 5 , 6 in that order.

An **in-order** traversal first visits the left subtree, then the root, then the right subtree. For the example tree, the visiting order is 3 , $+$, 4 , \times , 5 , $-$, 6 . The in-order traversal generates the usual *infix notation* of expressions (operator between operands) but without the brackets.

A **post-order** traversal first visits the left subtree, then the right subtree and finally the root. For the example tree, the visiting order is 3 , 4 , $+$, 5 , \times , 6 , $-$. This is the *postfix notation* that can be

evaluated with a stack.

All these traversals are forms of **depth-first search (DFS)**, a search that explores one subtree in depth before searching the other subtree.

16.3.2 Pre-order traversal

The algorithmic pattern for preorder(*tree*) is as follows, where step 2 is problem-dependent.

1. if *tree* is empty:
 1. stop
 2. visit root(*tree*)
 3. preorder(left(*tree*))
 4. preorder(right(*tree*))

A pre-order traversal can be used to print a tree. In this case, step 2 prints the root's value.

```
[2]: # this code is also in m269_tree.py

def write(tree: Tree, level: int) -> None:
    """Print the tree as in file browsers, with subtrees indented.

    Preconditions: level >= 0 is the initial indentation level
    """
    if is_empty(tree):
        print(" " * 4 * level, "EMPTY")
    else:
        print(" " * 4 * level, tree.root)
        write(tree.left, level + 1)
        write(tree.right, level + 1)
```

```
[3]: write(MTP, 0) # compare the output to the figure above
```

```
-  
 *  
 +  
 3  
   EMPTY  
   EMPTY  
 4  
   EMPTY  
   EMPTY  
 5  
   EMPTY  
   EMPTY  
 6
```

(continues on next page)

(continued from previous page)

```
EMPTY
EMPTY
```

The pre-order pattern may be modified for some problems. For example, when searching for an item in a tree, we can stop the traversal when we find it.

```
[4]: def has(tree: Tree, item: object) -> bool:
    """Return True if and only if the item occurs in the tree."""
    if is_empty(tree):
        return False
    if tree.root == item: # visit a node
        return True
    return has(tree.left, item) or has(tree.right, item)

has(MTP, 9)

[4]: False
```

This is similar to the recursive membership operation *on sequences*, which has two analogous base cases: the empty sequence and the head is the sought item.

The code uses short-circuit disjunction to search the right subtree only if the item isn't in the left subtree. In languages without short-circuiting we'd write:

```
if has(tree.left, item):
    return True
return has(tree.right, item)
```

Exhaustive search algorithms on trees usually do a pre-order traversal because by first looking at the root they may avoid visiting the subtrees.

16.3.3 In-order traversal

Other than the name of the recursive function, what do we need to change in the pre-order pattern to obtain in-order traversal?

We just need to swap steps 2 and 3 to visit the left subtree before the root.

A plain in-order traversal of an expression tree produces the expression in infix notation but without the parentheses. This is wrong because the original expression $(3+4)\times 5-6$ becomes $3+4\times 5-6$ which, according to the usual precedence of operators, means $3+(4\times 5)-6$. We have to print each subtree within brackets.

```
[5]: def infix(expression: Tree) -> None:
    """Print infix form of expression, with full brackets."""
    if is_empty(expression):
        return
```

(continues on next page)

(continued from previous page)

```
print("(", end="")
infix(expression.left)
print(" ", expression.root, " ", end="")
infix(expression.right)
print(")", end="")  
  
infix(MTP)  
( (( ( 3 ) + ( 4 )) * ( 5 )) - ( 6 ))
```

When the `print` function has argument `end=s`, it prints string `s` instead of the newline character.

Exercise 16.3.1

Here's the function again with some more tests. Change it so that brackets around integer literals aren't printed.

```
[6]: def infix(expression: Tree) -> None:
    """Print infix form of expression, with full brackets."""
    if is_empty(expression):
        return
    print("(")
    infix(expression.left)
    print(" ", expression.root, " ", end="")
    infix(expression.right)
    print(")", end="")  
  
infix(MTP)    # ((3+4)*5)-6
print()
infix(TPM)    # (3+4)*(5-6)
print()
infix(PMT)    # 3+((4*5)-6)
print()
infix(MPT)    # (3+(4*5))-6
```

Hint Answer

16.3.4 Post-order traversal

Other than the name of the recursive function, what do we need to change to the pre-order pattern to obtain post-order traversal?

We must move step 2 (visit the root) to the end of the algorithm, after visiting the right subtree.

We can evaluate an expression tree with a post-order traversal, because we can only process an operator after evaluating the left and right operands. The base case isn't the empty tree as that has no defined value; it's a leaf (a literal).

```
[7]: def evaluate(expression: Tree) -> int:
    """Return the value of the expression tree.

    Preconditions:
    - expression isn't empty
    - expression only has operators +, -, * and numeric operands
    """
    if is_leaf(expression):
        return expression.root
    left_value = evaluate(expression.left)
    right_value = evaluate(expression.right)
    operator = expression.root
    if operator == "+":
        return left_value + right_value
    if operator == "-":
        return left_value - right_value
    if operator == "*":
        return left_value * right_value

    infix(MTP)  # ((3+4)*5)-6
    print(" =", evaluate(MTP))
    infix(TPM)  # (3+4)*(5-6)
    print(" =", evaluate(TPM))
    infix(PMT)  # 3+((4*5)-6)
    print(" =", evaluate(PMT))
    infix(MPT)  # (3+(4*5))-6
    print(" =", evaluate(MPT))

    ((( 3 ) + ( 4 )) * ( 5 )) - ( 6 ) = 29
    ((( 3 ) + ( 4 )) * (( 5 )) - ( 6 )) = -7
    (( 3 ) + ((( 4 ) * ( 5 )) - ( 6 ))) = 17
    ((( 3 ) + (( 4 ) * ( 5 ))) - ( 6 )) = 17
```

If you run this cell after doing the previous exercise you get fewer brackets.

16.3.5 Breadth-first search

A **level-order** traversal goes through the tree level by level, from left to right within each level. This is a form of **breadth-first search (BFS)** because it goes through the breadth of each level before moving down to the next one.

Depth-first search uses the call stack of the interpreter for the recursive calls; breath-first search uses a queue. When we visit a node, we enqueue its children, i.e. the next level, to visit them later. As we visit each level from left to right and enqueue its children, the next level will also be

traversed left to right. The front of the queue is the next node to be visited. When we reach a leaf, no children are enqueued. At some point the queue will be empty and the algorithm stops. Here's the BFS pattern:

1. let *to visit* be the empty queue
2. enqueue *tree* in *to visit*
3. while *to visit* isn't empty:
 1. dequeue *tree* from *to visit*
 2. visit *root(tree)*
 3. if *left(tree)* isn't empty:
 1. enqueue *left(tree)* in *to visit*
 4. if *right(tree)* isn't empty:
 1. enqueue *right(tree)* in *to visit*

Here's a version of level-order traversal that prints one level per line. It uses two queues, one for the current level and the other for the next level, to detect when the current level ends and print a newline.

```
[8]: %run -i ../m269_queue
```

```
def levels(tree: Tree) -> None:
    """Print the tree from the root down, one level per line."""
    this_level = Queue()
    next_level = Queue()
    this_level.enqueue(tree)
    while this_level.size() > 0:
        tree = this_level.dequeue()
        print(tree.root, " ", end="")
        if not is_empty(tree.left):
            next_level.enqueue(tree.left)
        if not is_empty(tree.right):
            next_level.enqueue(tree.right)
        # if it was last tree of this level, start new line and level
        if this_level.size() == 0:
            print()
            this_level = next_level
            next_level = Queue()

levels(MTP)  # (3+4) * 5 - 6
-
* 6
```

(continues on next page)

(continued from previous page)

```
+ 5
3 4
```

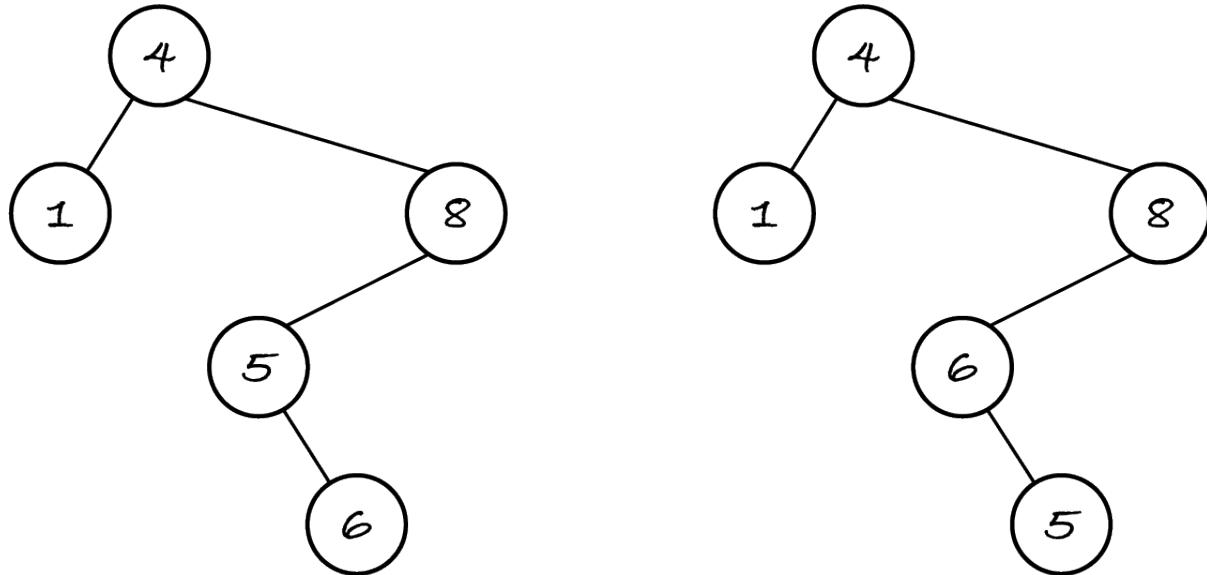
```
[9]: levels(TPM)  # (3+4) * (5-6)

*
+
-
3 4 5 6
```

16.4 Binary search trees

A **binary search tree (BST)** is a sorted binary tree, which allows us to find items using binary search instead of exhaustive search. In a BST, all items in the left subtree come before the root item, which in turn comes before all items in the right subtree. (This is the **ordering property** of BSTs.) What ‘comes before’ means depends on the problem at hand. Both subtrees are BSTs too, so that all items are sorted. The next figure shows two binary trees with integers.

Figure 16.4.1



If ‘comes before’ means ‘less than’, then the tree on the left is a BST and the other one isn’t because 5 is less than 6 but isn’t in the left subtree of 6. If ‘comes before’ means ‘greater than’, then neither tree is a BST.

If the items in a BST are key–value pairs and the keys are unique, then a BST can be used as the data structure for the *map ADT*. The keys must be comparable, otherwise the binary tree wouldn’t be a BST. Like in the chapter on sorting, I’ll keep BST items in ascending order of keys. A descending order only requires reversing the comparison operator in the following algorithms.

We’ll use the same `Tree` class as before, but each item will be a key–value pair, represented by a tuple. To make the code more readable I define two constants to index the tuple.

```
[1]: %run -i ./m269_tree
```

```
KEY = 0
VALUE = 1
```

I use the left-hand tree in Figure 16.4.1 to test the map operations. The values are irrelevant for the map operations so I'll keep drawing BSTs with the keys only, but I need to add values for the algorithms to work. To keep the example short, the BST will map Arabic to Roman numerals.

```
[2]: I = (1, "I")
IV = (4, "IV")
V = (5, "V")
VI = (6, "VI")
VIII = (8, "VIII")
```

```
one = join(I, Tree(), Tree())
six = join(VI, Tree(), Tree())
bst = join(IV, one, join(VIII, join(V, Tree(), six), Tree()))
```

16.4.1 Search

Since a BST is sorted, a binary search, with the root item being the equivalent of the middle item in sequences, can be used for two map operations: membership (check if a key exists) and lookup (get the value for a given key).

To check if a given key exists, we compare it to the root's key. If the keys differ, the binary search proceeds recursively on the left or right subtree depending on whether the given key is smaller or larger than the root's key, respectively.

```
[3]: def has(tree: Tree, key: object) -> bool:
    """Return True if and only if a node of tree has the key.
```

Preconditions: tree is a BST

"""

```
if is_empty(tree):
    return False
elif tree.root[KEY] == key:
    return True
elif tree.root[KEY] < key:
    return has(tree.right, key)
else:
    return has(tree.left, key)
```

```
has(bst, 2)
```

```
[3]: False
```

[4]: has(bst, 6)

[4]: True

Like binary search on sequences, binary search on BSTs is a tail-recursive decrease-and-conquer algorithm with two base cases. However, whereas binary search on sequences always reduces the search space by half, binary search on BSTs reduces it by a variable amount: the left and right subtrees may not have the same size, as the examples show. I'll look at the complexity of binary search on BSTs in the next section.

Similar to what happened with hash tables, another map ADT implementation, if we directly access and modify a key stored in the BST, we may break the ordering property and the membership operation won't find it anymore. For example, if key 6 is replaced with key 10, the tree is no longer a BST: the algorithm will search for 10 in the right subtree of 8 and not find it.

Exercise 16.4.1

Implement the lookup operation.

[5]: from algoesup import test

```
def lookup(tree: Tree, key: object) -> object:
    """Return the value associated to the key.

    Preconditions: tree is a BST and has the key
    """
    pass

lookup_tests = [
    ('key in leaf', bst, 1, 'I'),
    ('key in root', bst, 4, 'IV'),
    ('key in other', bst, 5, 'V')
]
test(lookup, lookup_tests)
```

Hint Answer

Exercise 16.4.2

Does any DFS or BFS traversal of a BST produce the items in ascending key order?

Hint Answer

Exercise 16.4.3

In a BST, the first and last items in sorted order are in the left- and right-most nodes, respectively. For example, the smallest key in the example is 1 because one can't go further left from that node and the largest key is 8 because one can't go further right from that node.

Implement the following function recursively or iteratively. Add any necessary preconditions. Run both tests.

```
[6]: def smallest(tree: Tree) -> object:  
    """Return the item in the tree with the smallest key.  
  
    Preconditions: tree is a non-empty BST  
    """  
    pass
```

```
[7]: smallest(bst) == I
```

```
[8]: smallest(six) == VI # this tree has a single node
```

Hint Answer

16.4.2 Add node

The map ADT's associate operation adds a new key–value pair or replaces the existing value.

Adding a node requires finding where it should be and adding it there. Here's a visualisation of how a BST is created from an unsorted sequence, by adding one item at a time. First we find where the key should be, using binary search. If we reach an empty subtree, we put a new leaf there with the given item. If we reach an existing node, we replace its value.

```
[9]: def associate(tree: Tree, key: object, value: object) -> None:  
    """Associate the value to the key in the tree.  
  
    Preconditions: tree is a BST  
    Postconditions:  
        - if there's a node with the key, replace its value with the  
        →given one  
        - otherwise, add the key-value pair to the tree  
    """  
    # Base case: if tree is empty, create a leaf  
    if is_empty(tree):  
        tree.root = (key, value)  
        tree.left = Tree()  
        tree.right = Tree()  
    # Base case: if the key is in the root, replace the value  
    elif tree.root[KEY] == key:  
        tree.root = (key, value)  
    # Recurrence relation: add/replace in the appropriate subtree
```

(continues on next page)

(continued from previous page)

```

elif tree.root [KEY] < key:
    associate(tree.right, key, value)
else:
    associate(tree.left, key, value)
    
```

Like for linked lists, inserting a node has the same complexity as searching: the insertion itself takes constant time once the place is found because no items are shifted, unlike inserting in arrays. Let's test the operation.

```
[10]: write(bst, 0)  # this is the left tree in Figure 16.4.1

(4, 'IV')
(1, 'I')
    EMPTY
    EMPTY
(8, 'VIII')
(5, 'V')
    EMPTY
(6, 'VI')
    EMPTY
    EMPTY
    EMPTY
```

Adding a node with key 2 will put it as the right child of 1.

```
[11]: associate(bst, 2, "II")
write(bst, 0)

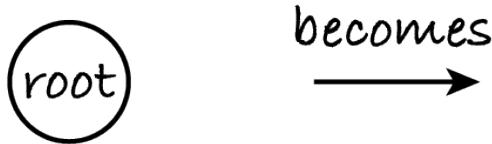
(4, 'IV')
(1, 'I')
    EMPTY
(2, 'II')
    EMPTY
    EMPTY
(8, 'VIII')
(5, 'V')
    EMPTY
(6, 'VI')
    EMPTY
    EMPTY
    EMPTY
```

16.4.3 Remove node

The map operation to delete an item has to first search for its key. At that point, we have a tree with a left subtree L, a right subtree R and the item to be removed in the root. We have to replace this old BST by a new BST without the root. We can't simply remove the root: that would lead to two disconnected subtrees. We must think what to do case by case.

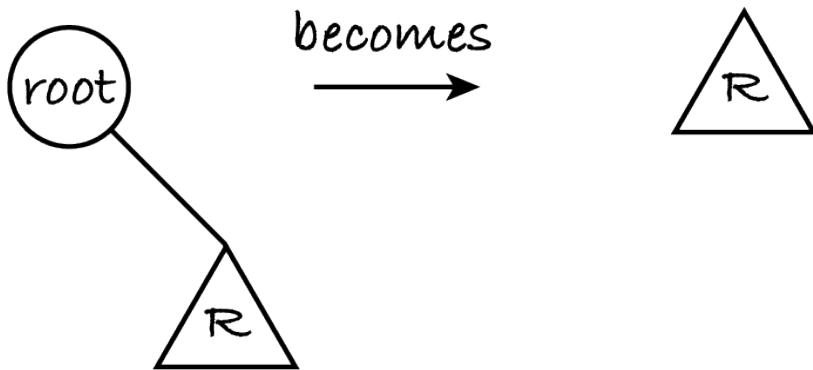
If the root is a leaf, i.e. L and R are empty, then the new tree is empty.

Figure 16.4.2



If L is empty, then the new tree is simply R, and vice versa.

Figure 16.4.3



Note that if R is empty, we have the same situation as for removing a leaf, so we can treat all these cases equally: if one subtree is empty, the new tree is the other subtree.

Finally, consider the case where neither subtree is empty. We must replace the old root by a node that keeps the ordering property: the key must come after the keys in the left subtree but before those in the right subtree. There are only two possible choices: the **predecessor** node, which has the key that comes immediately before the root's key and the **successor** node, with the key immediately after the root's key. If you think of a sorted sequence like (1, 2, 5, 6, 9), and we want to remove the 5, the only numbers that can take its place are 2, the predecessor of 5, or 6, the successor of 5. We cannot put any other number in place of the 5 as that would break the sorted order.

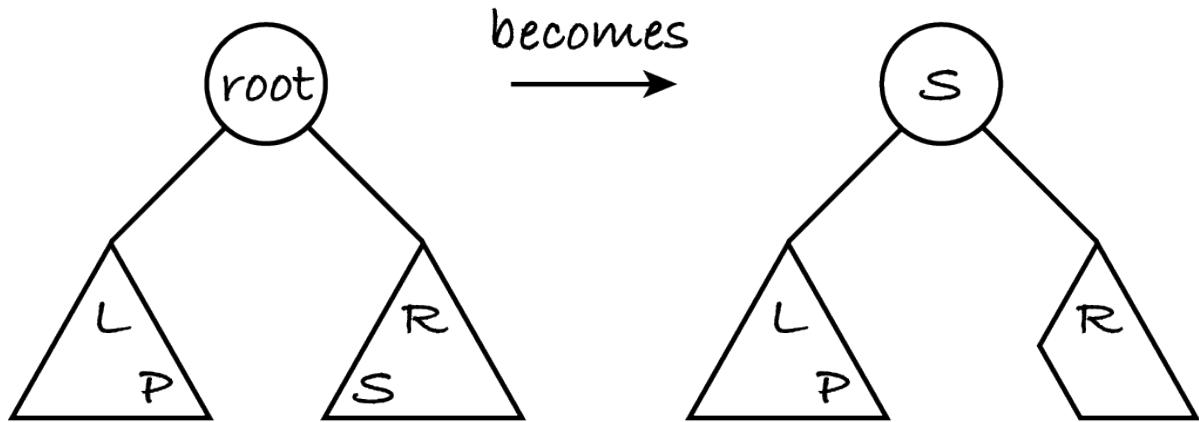
Where in the tree is the predecessor of the root? Alternatively, where's the successor?

The predecessor of the root is in the left subtree, by definition of a BST. Since the predecessor comes immediately before the root, it has the largest key in the left subtree, so it's the right-most node of the left subtree. Vice versa, the successor is the left-most node in the right subtree.

After putting the predecessor (or the successor) in place of the root, we have to remove it from its subtree, making a recursive call.

You already implemented the operation to obtain the smallest item in a tree (what a coincidence!), so we'll use it to determine the root's successor. The next figure shows the successor s replacing the root, while the predecessor p stays in its right-most place of the left subtree.

Figure 16.4.4



The successor has no left child, otherwise it wouldn't be the smallest key in the right subtree. Therefore, removing the successor falls into the base case above (one subtree is empty) and takes constant time (once we find it). The same applies if we had decided to replace the root by its predecessor, which has no right child.

To sum up, removing a node does three searches in the worst case: find the node to be removed and find its successor (or predecessor) twice, first to copy it to the root, then to remove it. So deletion also has the same complexity as searching. I'll come back to this in the next section.

```
[12]: def smallest(tree: Tree) -> object:
    """Return the item in the tree with the smallest key.

    Preconditions: tree is a non-empty BST
    """
    while not is_empty(tree.left):
        tree = tree.left
    return tree.root

def remove(tree: Tree, key: object) -> None:
    """Remove the tree's node with the key.

    Do nothing if there's no such node.
    """
    if is_empty(tree):
        pass # key not found
    elif tree.root[KEY] < key:
        remove(tree.right, key)
    elif key < tree.root[KEY]:
        remove(tree.left, key)
    else: # key found: it's in the root
        if is_empty(tree.left): # replace tree with right subtree
            tree.root = tree.right.root
            tree.left = tree.right.left
            tree.right = tree.right.right
        elif is_empty(tree.right): # replace tree with left subtree
            tree.root = tree.left.root
            tree.left = tree.left.left
            tree.right = tree.left.right
```

(continues on next page)

(continued from previous page)

```
tree.root = tree.left.root
tree.right = tree.left.right # note different order
tree.left = tree.left.left # of assignments
else: # replace root with successor
    tree.root = smallest(tree.right)
    remove(tree.right, tree.root[KEY])
```

Let's remove the node that was added above:

```
[13]: remove(bst, 2)
write(bst, 0)

(4, 'IV')
(1, 'I')
    EMPTY
    EMPTY
(8, 'VIII')
(5, 'V')
    EMPTY
    (6, 'VI')
        EMPTY
        EMPTY
    EMPTY
```

If we remove the root 4, it's replaced by its successor 5, which in turn 'promotes' its right subtree, with root 6, one level up:

```
[14]: remove(bst, 4)
write(bst, 0)

(5, 'V')
(1, 'I')
    EMPTY
    EMPTY
(8, 'VIII')
(6, 'VI')
    EMPTY
    EMPTY
    EMPTY
```

16.5 Balanced trees

As seen in the previous section, all the operations on a BST that aren't traversals rely on search and some constant-time operations. So the complexity of adding or removing a node, or replacing a value, is the complexity of binary search over a BST.

16.5.1 Complexity of search

A binary search takes constant time for each call. What's the best-case complexity of binary search on trees?

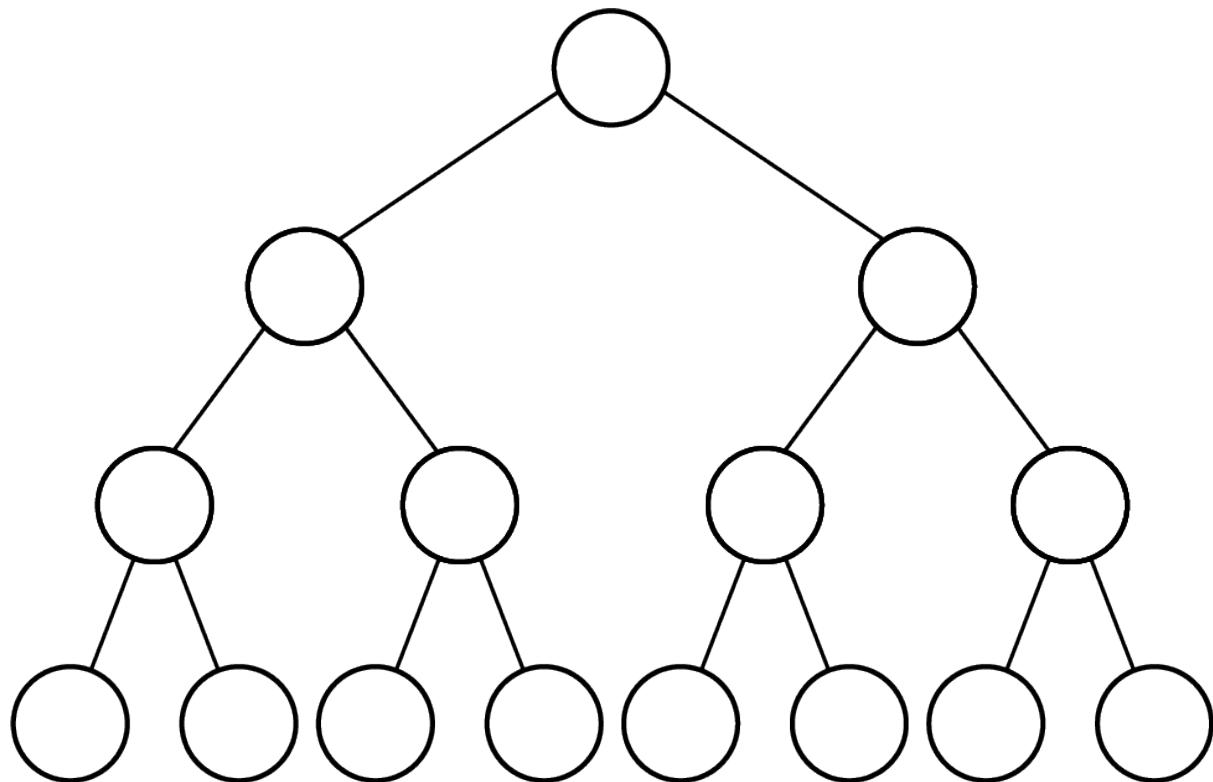
In the best case, the item sought is in the root and the search does one call: the complexity is $\Theta(1)$. Remember that a best- or worst-case scenario applies to varying input sizes, so the empty tree isn't a best-case scenario.

In the worst case, the search will go all the way down to the deepest leaf, where the sought item may be. Such a search will do one recursive call per level: the worst-case complexity is $\Theta(\text{height}(\text{tree}))$.

The height is maximal when the tree degenerates to a linked list, with each node having a single left or right child. In that case, each level has a single node, the height is equal to the size and the search has complexity $\Theta(|\text{tree}|)$.

The height is minimal when each level (except possibly the last one) is full, i.e. has as many nodes as possible. A perfect tree has minimal height for its size because all levels are full. Here's the shape of a perfect tree with four levels.

Figure 16.5.1



Each level has double the nodes of the previous level because each parent has two children. The number of nodes thus grows exponentially with the height: it can be shown that a perfect tree of height h has $2^h - 1$ nodes. This means that the minimal height for a tree of size n is $\log_2 n$. So, binary search on a perfect tree has complexity $\Theta(\log |\text{tree}|)$ in the worst case, when the sought item is in the lowest level.

It has been proven that if we add random comparable items one by one to an initially empty BST, then the height of the resulting BST is on average proportional to the logarithm of its size. Binary search thus has average complexity $\Theta(\log |\text{tree}|)$.

16.5.2 Balanced trees

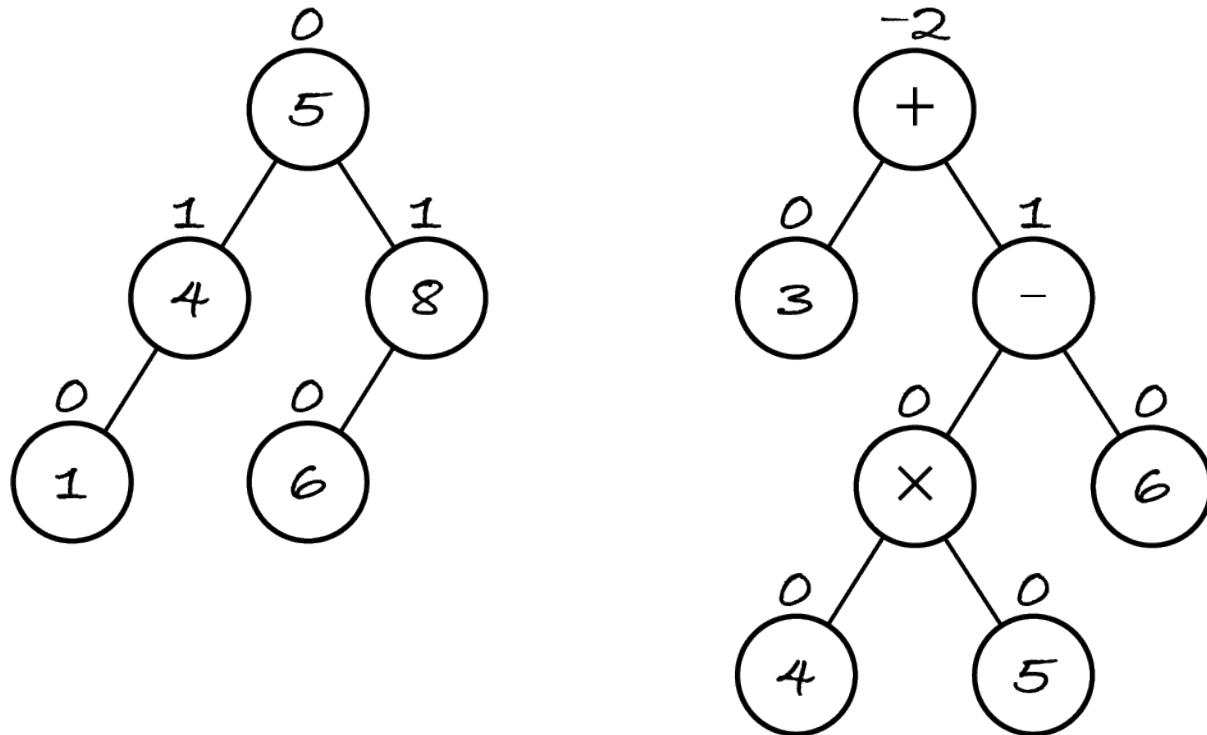
A perfect tree has minimal height because *every* node has subtrees of exactly the same height. No subtree is deeper than its sibling and so no subtree degenerates into a linked list.

Not every tree can have sibling subtrees with the same height. For example, a tree with only two nodes has necessarily one empty subtree of height 0 and one single-node subtree of height 1. To minimise the height of a binary tree we can only require subtrees to have as similar a height as possible. The precise definitions are as follows.

The **balance factor** of a node is the height of its left subtree minus the height of its right subtree, e.g. all leaves have balance factor 0. A binary tree is **balanced** if every node has balance factor $-1, 0$ or 1 . In other words, for every node of a balanced tree, its subtrees differ in height by at most one. An empty tree is balanced. A perfect tree is perfectly balanced: all nodes have balance factor 0.

The next figure shows two trees you've seen before; each node is annotated with its balance factor. The left-hand tree is balanced because all balance factors are 0 or 1. For example, the nodes in the middle level have a left subtree of height 1 and an empty right subtree so their balance factor is $1 - 0 = 1$. The other tree in the figure isn't balanced. The root node has a left subtree of height 1 and a right subtree of height 3 so the balance factor is -2 .

Figure 16.5.2



Even though the heights of sibling subtrees are not always equal, it can still be shown that the height of a balanced tree is proportional to the logarithm of its size, which is the best we can

expect. A **self-balancing BST** checks after each node addition or removal if it has become unbalanced and, if so, restores the balance. This ensures that binary search (and thus every map operation) always takes logarithmic time in the worst case.

There are several self-balancing BST data structures, including AVL trees, which restore the balance by rotating subtrees left or right to decrease the height of one and increase the height of the other. You can watch a [visualisation of an AVL tree](#) but in M269 you're not expected to know how an AVL tree works. The visualisation uses -1 instead of 0 as the height of an empty tree. The resulting balance factors are the same, because they're the difference of two heights.



Info: M250 Unit 10 introduces Java classes `TreeSet` and `TreeMap` in package `java.util`. They implement the set and map ADTs with self-balancing BSTs.

Exercise 16.5.1

1. If both subtrees of a non-empty tree are balanced, so is the tree. True or false?
2. If a non-empty tree is balanced, so are both its subtrees. True or false?

Hint Answer

16.5.3 Checking balance

To decide if a tree is balanced we need to check, for each node, if its balance factor is valid (-1, 0 or 1) and if both subtrees are balanced. Any traversal will do. I choose a pre-order traversal: the root is processed, i.e. its balance factor is computed, before traversing the subtrees.

```
[1]: %run -i ../m269_tree
from algoesup import test

def height(tree: Tree) -> int:
    """Return how many levels the tree has."""
    if is_empty(tree):
        return 0
    else:
        return max(height(tree.left), height(tree.right)) + 1

def is_balanced(tree: Tree) -> bool:
    """Return True if and only if the tree is balanced."""
    if is_empty(tree):
        return True
    else:
        valid_factor = -1 <= height(tree.left) - height(tree.right)
        ↵<= 1
```

(continues on next page)

(continued from previous page)

```

left_balanced = is_balanced(tree.left)
right_balanced = is_balanced(tree.right)
return valid_factor and left_balanced and right_balanced

is_balanced_tests = [
    ('empty tree', Tree(), True),
    ('leaf', SIX, True),
    ('unbalanced', PMT, False),
    ('balanced', TPM, True)
]

test(is_balanced, is_balanced_tests)

Testing is_balanced...
Tests finished: 4 passed (100%), 0 failed.

```

Let's first analyse the complexity of `height`. The worst-case scenario is a degenerate tree with one node per level, i.e. with height equal to the size. In that case, one subtree is empty and the other has all the remaining nodes. The divide-and-conquer approach degenerates to a decrease-by-one approach. If n is the size of the tree, then the complexity of the `height` function is:

- $T(0)$ to recursively handle the empty subtree
- $T(n - 1)$ to recursively handle the other subtree
- $\Theta(1)$ to take the largest value and add one.

The base case takes constant time, so the recursive definition of T is:

- if $n = 0$: $T(0) = \Theta(1)$
- if $n > 0$: $T(n) = T(n - 1) + T(0) + \Theta(1) = T(n - 1) + \Theta(1)$.

This leads to $T(n) = \Theta(n)$.

Exercise 16.5.2

Define the worst-case complexity for `is_balanced` recursively. What is the resulting complexity?

Hint Answer

The best-case scenario is the perfect tree, which has minimal height and equally divided nodes among the subtrees. The complexity is defined as above, but each recursive call now takes $T(n / 2)$ time. So the best-case complexity for the `height` function is:

- $T(0) = \Theta(1)$
- $T(n) = T(n / 2) + T(n / 2) + \Theta(1) = 2 \times T(n / 2) + \Theta(1)$.

This leads to $T(n) = \Theta(n)$, which fits our intuition. Computing the height of a tree requires visiting all nodes, no matter how they're distributed in the tree. Like the size operation, the height operation always takes linear time.

Having the best-case complexity of `height`, we can define the best-case complexity of `is_balanced`:

- $T(0) = \Theta(1)$
- $T(n) = 2 \times T(n/2) + \Theta(n / 2) + \Theta(n / 2) = 2 \times T(n/2) + \Theta(n)$.

This leads to $T(n) = \Theta(n \log n)$. Although the function is doing a linear time operation (computing the height) for each subtree, in a perfect tree each subtree has half the size, hence the logarithmic component.

Exercise 16.5.3

What changes would you make to the above algorithm to decide more efficiently whether a tree is balanced? You can describe the changes briefly instead of writing code.

Hint Answer

16.6 Heapsort

Selection sort searches the unsorted part for the item with the smallest key, removes it and appends it to sorted part. We can see an item's key as the priority of being chosen next and use a min-priority queue to retrieve the next item for the sorted part.

1. let *queue* be a min-priority queue
2. for each *item* in *unsorted*:
 1. enqueue *item* in *queue* with priority *key(item)*
 3. let *sorted* be the empty sequence
 4. while *queue* isn't empty:
 1. append front of *queue* to *sorted*
 2. dequeue *queue*

This is a form of selection sort with pre-processing steps 1 and 2 to put the unsorted part in a min-priority queue, which thereby sorts it.

If we implement the priority queue with a self-balancing BST, adding and removing an item in steps 2.1 and 4.2 takes logarithmic time. Selection sort on an array takes quadratic time to do n linear searches; with a BST-based min-priority queue, it takes log-linear time.

Pigeonhole sort is also a form of selection sort with a data structure that allows getting the lowest key item in constant time.



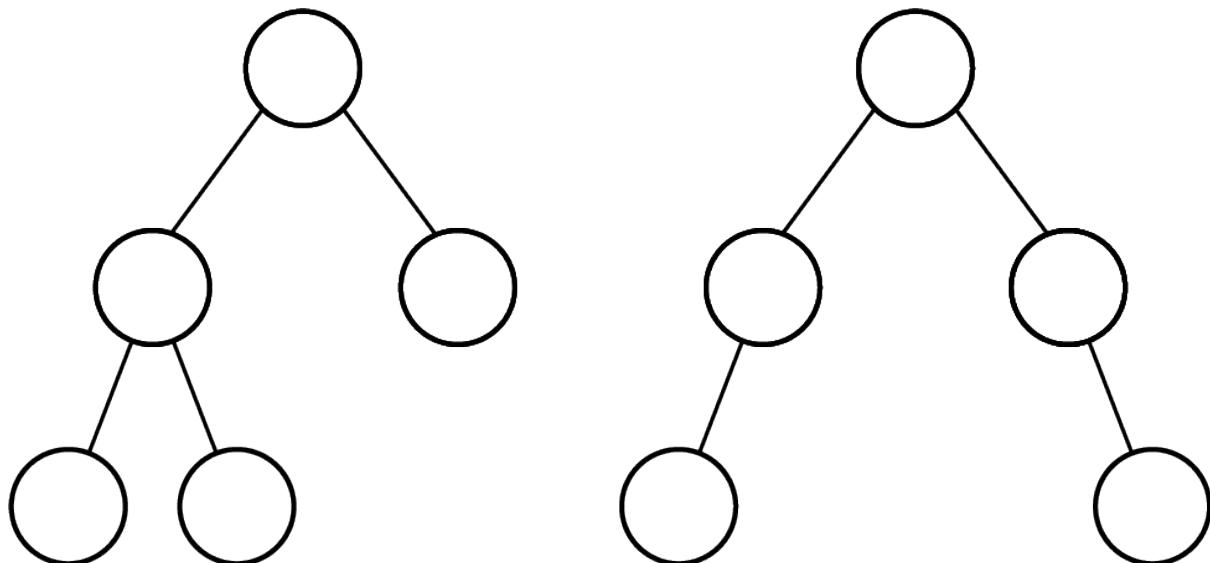
Note: An algorithm's complexity may be improved by using an auxiliary data structure.

Self-balancing BSTs have a memory overhead of two pointers per item and a run-time overhead to keep the tree balanced. There's a more efficient data structure for priority queues.

16.6.1 Binary heap

In a **complete** tree all levels are full except possibly the last one and there are no 'gaps' in the bottom-level leaves, i.e. they were added from left to right. A complete tree has minimal height and is balanced. The next figure shows two balanced binary trees. Only the left-hand tree is complete: the right-hand tree doesn't have middle leaves in the deepest level.

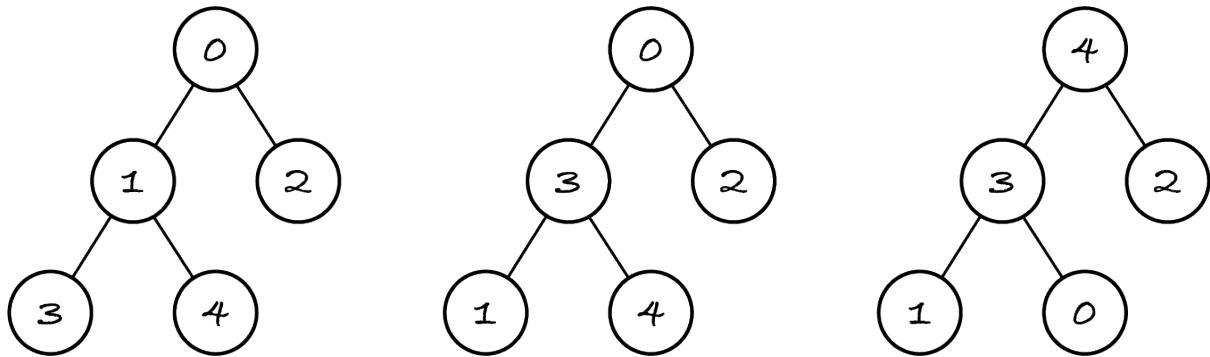
Figure 16.6.1



A **binary heap** is a complete binary tree with an ordering property: in a **min-heap** each node's key is less than or equal to its children's keys, whereas in a **max-heap** it's larger than or equal to. A min-heap has the smallest item in the root; a max-heap has the largest. In M269, heap means a binary heap, since we're not using other kinds of heaps.

The next figure shows a min-heap on the left, a max-heap on the right, whereas the middle tree is complete but not a heap: node 3 is smaller than one of its children but larger than the other.

Figure 16.6.2



Exercise 16.6.1

Can an item be searched in logarithmic time in a binary heap?

Hint Answer

A complete binary tree can be efficiently stored in an array in level order. We put the root in index 0, its left child in index 1, its right child in index 2, and so on. This way, the node at index i has its children at indices $2 \times i + 1$ and $2 \times i + 2$ and its parent at index $\text{floor}((i - 1) / 2)$. This is illustrated by the left-hand tree of the previous figure, where the keys are the same as the nodes' indices. For example, node 1 has children $2 \times 1 + 1 = 3$ and $2 \times 1 + 2 = 4$, and nodes 1 and 2 have parent $\text{floor}((1-1) / 2) = \text{floor} ((2-1) / 2) = 0$.

This approach is much more efficient in terms of memory and run-time than a pointer-based implementation of a complete tree. The latter requires three pointers per node to also access a node's parent, whereas the array-based implementation multiplies and divides indices by 2, which is very fast in binary arithmetic.

Heapsort is the above min-priority queue version of selection sort, using a min-heap as the data structure. Before I go into the details of the heap operations, you should see them and heapsort at work in this [visualisation](#). (Ignore the references to Miller and Ranum's textbook.)

16.6.2 Inserting items

To add an item to a min-heap, we first put it in the next available leaf position, i.e. we append it to the array. This guarantees the tree remains complete. However, the new item may be smaller than its parent. If that's the case, we swap it with its parent to satisfy the heap's ordering property. If the new item is still smaller than its new parent, we keep swapping it up the tree until it becomes the root or is the child of a smaller item.

The algorithm keeps two variables with the current index of the new item and its current parent, and updates them as the item 'bubbles' up the tree.

1. append *item* to *heap*
2. let *index* be $|heap| - 1$
3. let *parent* be $\text{floor}((index - 1) / 2)$
4. while $index \neq 0$ and $\text{key}(item) < \text{key}(heap[parent])$:
 1. swap $heap[index]$ with $heap[parent]$
 2. let *index* be *parent*

3. let $parent$ be $\text{floor}((index - 1) / 2)$

The algorithm relies on short-circuit conjunction in step 4, so that the invalid parent index isn't accessed when the item becomes the root.

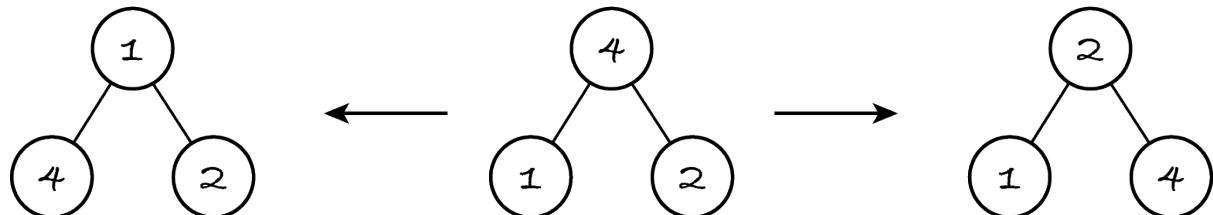
16.6.3 Removing the root

The only item that can be removed from a min-heap is the root, which is the front of the priority queue. When the root is removed the last leaf node replaces it, so that the tree remains complete. However, this will break the ordering property: the old root was the smallest item but the new root is one of the largest items, because it was in the lowest level. To reinstate the ordering, we simply swap the new root with one of its children and keep swapping it down until it becomes a leaf again or is smaller than its children.

Promoting one of the largest items to the root only to move it back down sounds daft, but the actual point is to move the appropriate smaller items up as the swaps are made.

Does it matter which child the parent is swapped with? Let's think with a concrete example, the min-heap of Figure 16.6.2. If we dequeue the root, it gets replaced by the last leaf with key 4. The middle of the next figure shows the new root and its children.

Figure 16.6.3



If we swap node 4 with its right child, as shown in the right-hand side of the figure, it still isn't a min-heap, because the new root, with key 2, is larger than the left child. If we instead swap node 4 with its left child, node 1, as shown in the left-hand side of the figure, we reinstate the ordering property.

In general, we must swap the root with the smallest of its children. This guarantees the child being promoted to parent is smaller than its two new children: its former parent and its former sibling.

The algorithm to dequeue the root is as follows, with $front$ being the output variable:

1. let $front$ be $heap[0]$
2. let $last$ be $|heap| - 1$
3. let $heap[0]$ be $heap[last]$
4. remove $heap[last]$
5. $\text{down}(heap, 0)$

The auxiliary function $\text{down}(heap, index)$ moves the node at the given index down as far as possible. The easiest implementation is recursive. One base case is that the node is a leaf, the other that it's already smaller than its children. In both cases, nothing is done. Otherwise the node is swapped with the smallest child and pushed down further.

1. let $left$ be $2 \times index + 1$
2. if $left \geq |heap|$:
 1. stop
3. let $right$ be $left + 1$
4. if $right < |heap|$ and $key(heap[right]) < key(heap[left])$
 1. let $smallest$ be $right$
5. otherwise:
 1. let $smallest$ be $left$
6. if $key(heap[index]) > key(heap[smallest])$:
 1. swap $heap[index]$ with $heap[smallest]$
 2. down($heap, smallest$)

Step 2 checks that the node to be pushed down is a leaf by checking if the index of its left child is out of bounds. Step 4 uses short-circuit conjunction to first check if there's a right child before comparing it to the left child.

16.6.4 Complexity

The worst-case scenario for inserting an item is to bubble it up from the bottom level to the root. Likewise the worst-case scenario for dequeuing the root is to push the new root (which was the right-most leaf) down back to the bottom level.

Exercise 16.6.2

What's the worst-case complexity of heapsort?

Hint Answer

Exercise 16.6.3

What's the best-case scenario and complexity of heapsort?

Hint Answer

In summary, their weaker ordering property than for BSTs and their structural property (being complete trees) makes heaps more efficient for implementing priority queues than self-balancing BSTs: although each priority queue operation takes logarithmic time for both data structures, heaps use less memory and execute operations faster.

Despite heapsort having better worst-case complexity than quicksort, in practice quicksort is preferred because heapsort accesses all parts of the array as it swaps items. If the array doesn't fit in cache memory, then the various parts of the array are copied in and out of the cache as the item bubbles up or down. In contrast, quicksort doesn't access any part of the array outside the partition it's working on. Every partition small enough to fit in the cache stays there until it's sorted. We say that quicksort has better cache locality than heapsort. A well-implemented in-place quicksort algorithm runs faster than heapsort for large arrays.



Info: TM112 Block 1 Section 3.1.2 explains cache memory.

16.6.5 Heaps in Python

Python's module `heapq` implements min-priority queues with min-heaps. No new data type is introduced: a heap is represented by a list. The module includes function `heappush(heap, item)` to add the given item to the given heap, and function `heappop(heap)` to remove and return the smallest item. Note that items are assumed to be directly comparable: there's no additional parameter for a key function.

To access the front of the priority queue without removing it, just write `heap[0]`. You can use any other list operation, but most of them are meaningless for priority queues and will break the heap properties.

Here's the heapsort algorithm, with a key function. I add each item to the heap as a key–item tuple. Due to lexicographic comparison, the keys are compared first, then the items, if they have the same key. This means that items must be comparable, contrary to the other sorting algorithms, where only keys are. The key is discarded before appending the item to the output sequence.

```
[1]: from typing import Callable
      from algoesup import test

%run -i ../m269_sorting

from heapq import heappush, heappop

def heapsorted(unsorted: list, key: Callable) -> list:
    """Return a permutation with keys in non-decreasing order.

    Preconditions:
    - all items in unsorted are pairwise comparable
    - for any indices i and j,
        key(unsorted[i]) and key(unsorted[j]) are comparable
    """
    heap = []
    for item in unsorted:
        heappush(heap, (key(item), item))
    result = []
    while len(heap) > 0:
        result.append(heappop(heap)[1])
    return result

test(heapsorted, sorting_tests)
```

```
Testing heapsorted...
Tests finished: 7 passed (100%), 0 failed.
```

Let's confirm that heapsort is log-linear.

```
[2]: from random import shuffle

for doubling in range(5):
    items = list(range(100 * 2**doubling, -1, -1))
    shuffle(items) # random sequence
    %timeit -r 5 heapsorted(items, identity)

22.9 µs ± 55.8 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴each)
48.7 µs ± 54 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴each)
105 µs ± 144 ns per loop (mean ± std. dev. of 5 runs, 10,000 loops
 ↴each)
225 µs ± 300 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↴each)
487 µs ± 584 ns per loop (mean ± std. dev. of 5 runs, 1,000 loops
 ↴each)
```

16.7 Summary

16.7.1 Rooted trees

A **rooted tree** consists of zero or more **nodes**. Each node contains an item and is the **parent** of zero or more **child** nodes. The root node has no parent; every other node has exactly one parent. The **leaves** are the nodes without children. The **size** of a tree is how many nodes (or items) it has.

Rooted trees represent hierarchical collections of items, organised by **levels**. The **height** of a tree is how many levels it has. Level zero, the top level, contains only the root. If a node is at level n , then its children are at level, or **depth**, $n + 1$ and its parent is at level $n - 1$.

The **ancestors** of a node are its parent, parent's parent, and so on until the root. The **descendants** of a node are its children, children's children and so on until reaching leaf nodes.

There are two basic ways of exhaustively searching a rooted tree. A breadth-first search (BFS) generates the nodes level by level: all of a node's children are tested before testing their children. A depth-first search (DFS) generates and tests all descendants of a node's child before doing the same for the next child of that node. A **pre- or post-order traversal** is a DFS that tests a node respectively before or after its children.

16.7.2 Binary trees

In a **binary tree**, each node has at most two children, called the **left child** and **right child**. A binary tree can be recursively defined as being empty or consisting of a root and two binary trees, called the left and right **subtrees**. The binary tree ADT operations are directly based on this definition:

Operation	Effect
new	create a new empty binary tree
join(i, l, r)	create a tree with root item i and subtrees l and r
root(t)	obtain the root item of binary tree t
left(t)	obtain the left subtree of t
right(t)	obtain the right subtree of t
t is empty	check if t has no nodes

When writing algorithms in English, the new operation is written as: let t be an empty binary tree. Operations root, left and right assume t isn't empty. All operations can be implemented in constant time.

Due to their recursive structure, binary trees can be processed with recursive divide-and-conquer algorithms. The base case is either an empty tree or a leaf. **Arm's-length recursion**, which tests the base case before the recursive call, should be avoided as it complicates the algorithm and usually makes it slower.

An **in-order traversal** of a binary tree is a DFS that processes the root in between processing the left and right subtrees.

The **balance factor** of a node is the difference between the left subtree's height and the right subtree's height. A binary tree is **balanced** if every node has balance factor $-1, 0$ or 1 .

A binary tree is **complete** if all levels, except possibly the last one, are full and the leaves on the last level have no gaps from left to right. A binary tree is **perfect** if all its levels are full. Complete trees and perfect trees are balanced.

The height of a binary tree is at least $\log |tree|$, for a complete tree, and at most $|tree|$, for a tree that has one node per level. For a balanced tree, the height is proportional to $\log |tree|$.

16.7.3 Binary search trees

A **binary search tree (BST)** is a binary tree with an ordering property: each item is larger than the items in the left subtree and smaller than the items in the right subtree. Moreover, each subtree is a BST itself. An in-order traversal of a BST produces items in ascending order.

BSTs are used to implement the map ADT, if items are key-value pairs with unique and comparable keys. All map operations require to first search for the node with the given key. This can be done with a binary search, due to the ordering of the items.

Binary search visits one node per level so it takes linear time in the height of the tree, assuming each visited node takes constant time to process. Binary search is $\Theta(1)$ in the best case, $\Theta(\log |tree|)$ in the average case, and $\Theta(|tree|)$ in the worst case.

A **self-balancing BST** automatically balances itself after a node has been inserted or removed.

16.7.4 Heaps

A **min-heap** or **max-heap** is a binary tree with an additional structural property (it's complete) and an ordering property: each item (or its key) is respectively smaller or larger than its children (or their keys). Min- and max-heaps are used to implement priority queues: the smallest or largest priority item is in the root.

A heap can be efficiently stored and manipulated in an array. A new item is added as the last node and bubbles up the tree while it's smaller (or larger, for a max-heap) than the parent. When the root is removed, the last item takes its place and bubbles down while it's larger (respectively, smaller) than their children. Inserting an item and removing the root take logarithmic time at worst, when the item moves up (or the new root moves down) the height of the tree.

In Python, min-heaps can be created by repeatedly calling `heappush(heap, item)` on an initially empty list `heap`. The items added to the heap must be comparable. Calling `heappop(heap)` on a non-empty heap returns and removes the root item. Both functions are in module `heapq`.

Heapsort is a form of selection sort with unsorted items kept in a min-heap instead of a sequence. Heapsort has linear best-case complexity and log-linear worst-case complexity. Due to swapping items up and down across the whole array, heapsort doesn't have good cache locality and is slower in practice than in-place quicksort.

CHAPTER 17

GRAPHS 1

In collections seen so far, either all items are related to each other by some ordering (sequences), no items are related to each other (sets), or some items are related by a hierarchical order (rooted trees).

This chapter introduces graphs, which explicitly state which pairs of items are related to each other (and, by omission, which aren't). Graphs can model all kinds of networks. For example, we can model a social network as a graph where the items represent people and two items (people) are related if and only if they're mutual friends.

Algorithms on graphs can then be used to recommend new friends to a person, suggest whom they should ask for an introduction to another person, target ads at popular people who can broadcast them, etc. All social media run on graphs.

Due to their flexibility and generality, graphs are one of the most widely used ADTs across many domains besides social media: transport and logistics, computer networks, biology, linguistics, etc.



Info: Like other terms, ‘graph’ has multiple meanings. In MU123 and MST124, a graph is an x-y plot, like those in Sections 11.2.3 and 11.4.3. In M269 and MST368, a graph is an abstract model of a network.

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn about the graph ADT and how to implement it.
- Develop and apply algorithms and data structures to solve computational problems – you will learn how to spot if a problem involves a graph.
- Analyse the complexity of algorithms to support software design choices - you will learn that the complexity of graph algorithms requires considering the number of items and the number of relations.

- Write readable, tested, documented and efficient Python code – this chapter introduces special graphs that serve as edge cases for testing graph algorithms.



Note: This is the second-longest chapter in the book. Don't feel rushed to complete it in one week. It's best to see this and the next chapter, which is shorter and has fewer exercises, as a two-week unit mostly about graphs.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

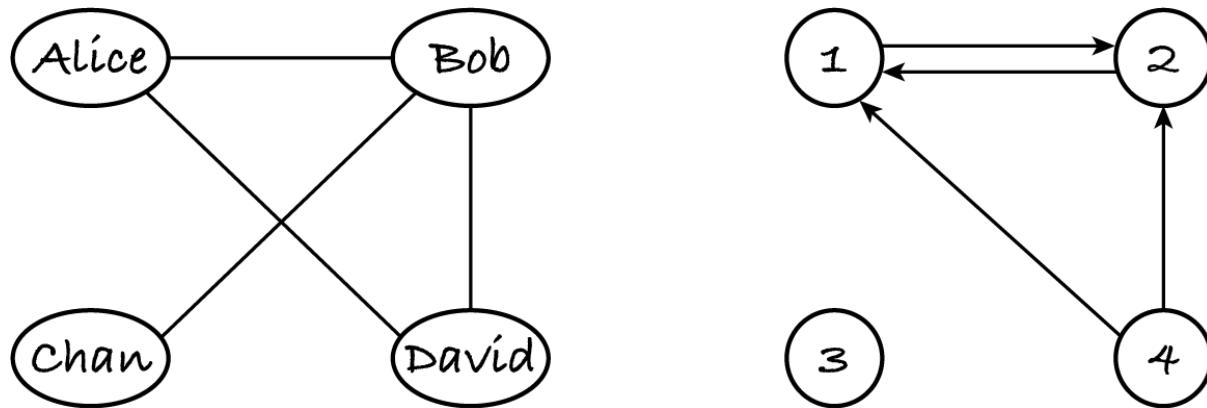
17.1 Modelling with graphs

Graphs are simple abstract mathematical structures and that's what makes them so powerful: they can model myriad real-world situations. Once you learn what graphs are and what they can represent, it's difficult not to see graphs everywhere.

An **undirected graph** consists of a set of items called **nodes** and a set of **undirected edges**, such that each edge connects two different nodes and there's at most one edge between any pair of nodes. A **directed graph** (abbreviated **digraph**) consists of a set of nodes and a set of **directed edges**, such that each edge connects two different nodes and there's at most two edges in opposite directions between any pair of nodes.

Graphs are usually depicted with the items within circles or ellipses and edges as lines. Directed edges have an arrowhead to show their direction. The next figure shows an undirected graph on the left and a digraph on the right.

Figure 17.1.1



Any undirected graph can be represented as a digraph. If we replace each undirected edge by two opposing directed edges, we transform an undirected graph into an equivalent digraph, albeit with double the number of edges.



Info: Nodes are also called vertices (singular: vertex) and edges are also called links.

Directed edges are also called arcs. There are many kinds of graphs; those defined above are called simple graphs.

As I mentioned in the introduction, graphs can represent all sorts of networks.

Several transport networks can be modelled by undirected graphs.

- A railway network can be modelled with nodes representing stations and edges representing track between stations. Track doesn't have a direction, it can be used by trains travelling in both directions, so the edges are undirected.
- A flight network can be modelled with nodes representing airports and edges representing the existence of a direct flight between two airports. Assuming there's a return flight between the same airports, the edges are undirected.
- The motorway network of a country can be represented by an undirected graph, because motorways can be used in both directions. Nodes represent junctions and edges represent stretches of road between them.

It's often better to use digraphs, in order to represent transport networks in more detail.

- The street network of a city can be modelled by a digraph with nodes representing junctions and edges representing streets. The direction of the edge indicates how the traffic flows. Directed edges allow us to distinguish one-way from two-way streets.
- If one direction of a motorway segment between two junctions is blocked, e.g. due to an accident, we need to represent each direction separately. A digraph with opposing edges between pairs of junctions is a more flexible model than an undirected graph, because it allows us to remove edges (and add them back) as the situation on the ground changes.

Like transport networks, communication and information networks can be modelled by undirected or directed graphs, depending on whether communication and information flows both ways between any pair of nodes or not, and on whether we want to represent each direction of flow separately or not.

The internet (a communication network) can be modelled as an undirected graph with nodes representing computers and edges representing a direct link (like an Ethernet cable or a Wi-Fi connection) between two computers. The World Wide Web (an information network) can be modelled as a digraph, where nodes represent web pages and an edge from A to B indicates that page A links to page B.

A spreadsheet can also be seen as an information network, with nodes representing cells and a directed edge from node A to node B if the value of cell A is used by the formula in cell B. Whenever the user changes the value of a cell, the spreadsheet app follows outgoing edges to quickly recompute only those formulas that directly or indirectly depend on that cell.

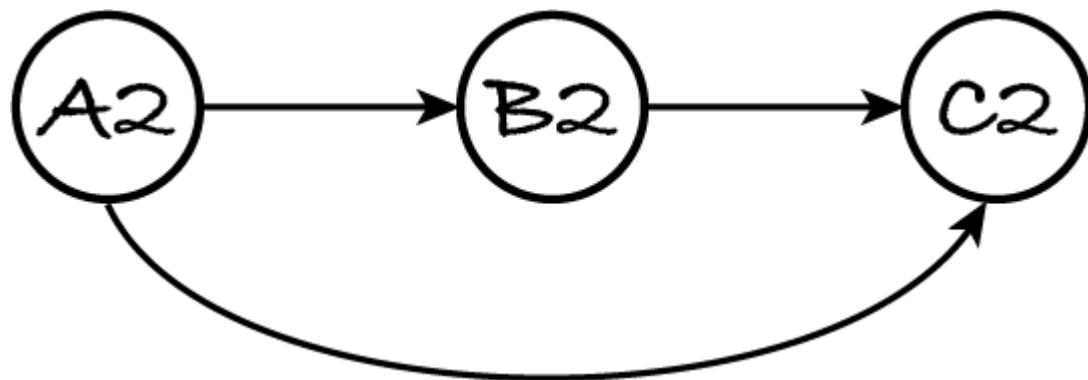
The following spreadsheet adds 20% VAT to the cost of a good or service.

Figure 17.1.2

	A	B	C
1	Cost	VAT	Total
2	5	=A2*0.2	=A2+B2
3			

The underlying digraph is:

Figure 17.1.3



Whenever the value in cell A2 changes, the app follows the outgoing edge to recompute B2 and then its outgoing edge to recompute C2.

Another example from Computing is the digraph formed by the data and the variable names in memory and the references between them. As the diagrams in [Section 6.2.1](#) show, the nodes are the variables and the data, and an edge from A to B indicates that A refers to B. An assignment $A = C$ removes any existing edge from A and creates an edge from A to the data referred to by C.

Biological networks are usually modelled with digraphs. For example, a food web is modelled with nodes representing animal and plant species, and an edge from node A to node B indicating that animal A eats animal or plant B.

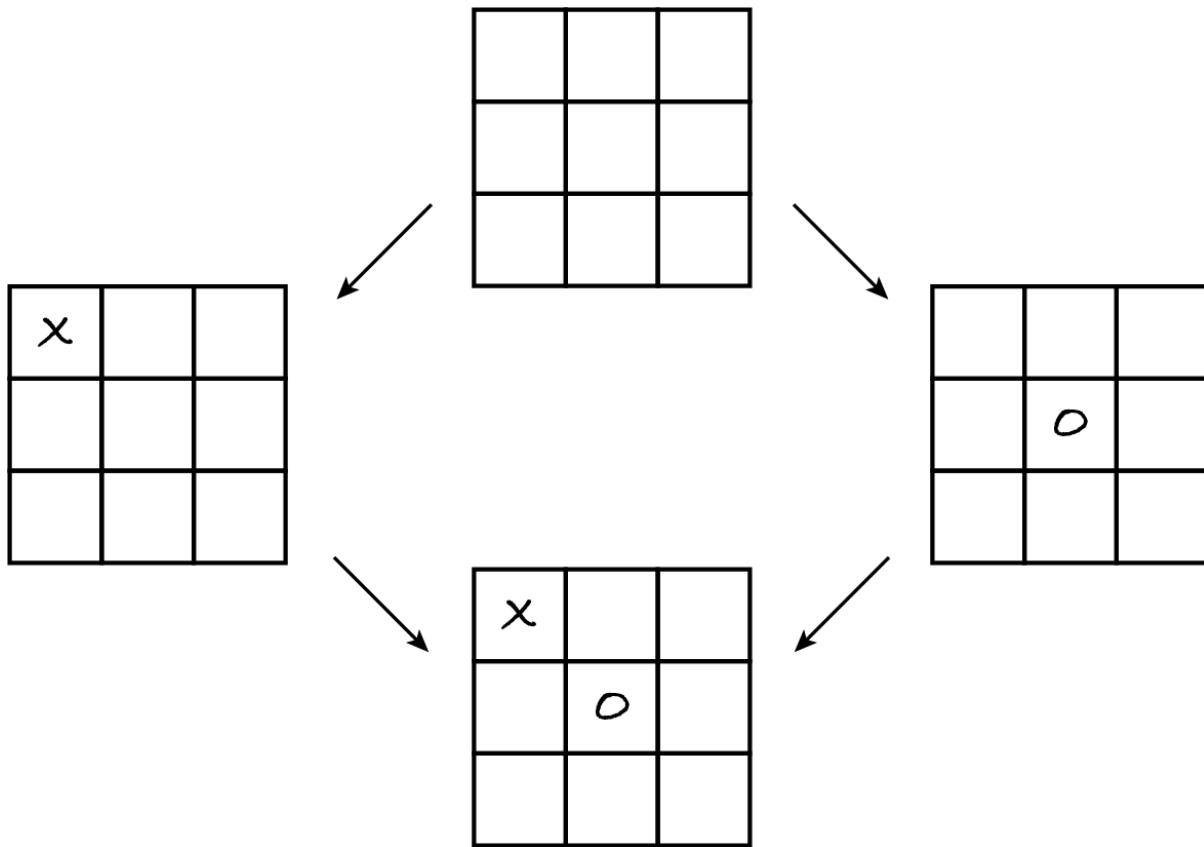
In a spreadsheet, a formula in one cell depends on the values in other cells; in a food web, a species depends on other species for nourishment. More generally, digraphs can represent networks of dependencies. For example, a digraph can model a work schedule, with nodes representing tasks and an edge from A to B indicating that task A has to be done before task B.

Social networks are modelled with graphs where nodes represent people. To model the Twitter network, we use a digraph with an edge from A to B indicating that A follows B. To model the Facebook or LinkedIn networks, we use an undirected graph with an edge between two people indicating that they are friends or acquainted.

Digraphs can also represent state transition diagrams. Each node is a possible state and an edge from A to B represents a transition: it's possible to go from state A to state B. For example, each possible state of a [Noughts and Crosses game](#) corresponds to a node and each edge to a valid

move, leading from one state to another. For example, if either player can start the game, the digraph includes the following nodes and edges (among many others).

Figure 17.1.4



Programs can then analyse the graph to determine a good move. Except for the simplest of games, the graph is too big to be fully analysed, so the program works on a partial graph: for example it generates only the states reachable from the current state in three moves. We'll return to state transition graphs in a *later chapter*.

In summary, whenever you want to represent a set of entities and a **binary relation**, i.e. between pairs of entities, a graph may be the best choice. If the relation is **symmetric**, i.e. holds in both directions, then you can use an undirected graph; otherwise it has to be directed. The ‘A is friends with B’ relation on Facebook is symmetric: if Alice is Bob’s friend, then Bob is Alice’s friend. The ‘A follows B’ relation on Twitter isn’t symmetric: Alice may follow Bob without Bob following Alice.

For asymmetric relations we often have a choice about the direction of edges. We can represent the same Twitter network with edges representing the ‘follows’ or the ‘is followed by’ relation, depending on the problem at hand. The ‘is followed by’ graph can be obtained from the ‘follows’ graph by reversing the direction of the edges.

Likewise, if there’s an edge from task A to task B in a work schedule graph this could either mean that task A depends on (and thus is done after) task B, or that task A prepares for (and thus is done before) task B.

When you’re asked to model some situation with a graph, you must state exactly what the edges and their direction represent. All edges must represent the same relation, otherwise the model is

useless.

Once you have modelled a set of entities and their relations as a graph, you can use general-purpose algorithms to process your graph. For example, you may have experienced a spreadsheet app telling you there's a circular dependency between your formulas. The app simply uses a general algorithm to find a cycle in any digraph. The same algorithm can find circular dependencies in a work schedule.

If a digraph has no cycles, there's an algorithm for sequencing the nodes so that for every edge from A to B, B will appear after A in the sequence. Such an algorithm is used by a spreadsheet to determine in which order to recalculate values: in the above example, cell C2 must be recomputed after B2. It's also used by *garbage collection*: if name or datum A refers to B which in turn refers to C, and nothing else refers to them, then A, B and C can be garbage collected, in that order.

As a final example, there's a general algorithm that finds the shortest path, with the least number of edges, to go from a node A to a node B. The same algorithm can be used on various networks to find, for example, a flight itinerary with the fewest transfers or a path for a data packet to go through the fewest intermediary computers.

No wonder graphs and their algorithms are so widely used, in games, satnavs, spreadsheets, language interpreters, by transport, communication and social media companies, by scientists studying how gossip and diseases spread, etc.

17.1.1 Exercises

Exercise 17.1.1

A train network has a north–south train running between stations N and S, stopping at stations A and B in between. There's also an east–west train that runs between stations E and W and stops at B in between.

Draw an undirected graph that represents direct travel: there's an edge between two stations if one can travel from one to the other without changing trains.

Don't worry too much about the layout of your graph: edges may cross, as in the undirected graph above.

Hint Answer

Exercise 17.1.2

Give two examples of relations between people (one symmetric, the other not) that don't involve social media.

Answer

Exercise 17.1.3

How could you represent a sequence of items (e.g. a queue) as a graph? State what the nodes and edges represent and whether the edges are directed.

Hint Answer

Exercise 17.1.4

How could you represent a set of items as a graph? State what the nodes and edges represent and whether the edges are directed.

Answer

Exercise 17.1.5

Some train and subway networks have several lines connecting two stops. For example, on the London Underground you can take the Circle or District line from Westminster to Tower Hill.

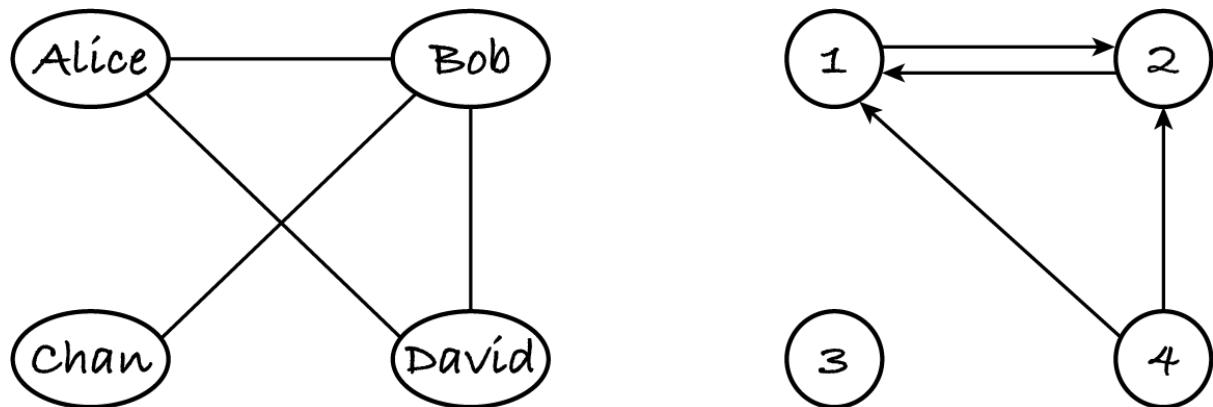
Our graph definitions don't allow multiple edges between two nodes. How could you model such networks then?

Hint Answer

17.2 Basic concepts

Before I introduce the graph ADT and associated data structures and algorithms, we need a bit more jargon, to facilitate communication. I use the same graphs as before to illustrate the concepts.

Figure 17.2.1



17.2.1 On nodes and edges

Two nodes are **adjacent** if they're connected by an edge. Nodes 2 and 4 are adjacent; 'Alice' and 'Chan' aren't. The **neighbours** of a node are all the nodes adjacent to it. The neighbours of node 1 are nodes 2 and 4.

In a digraph, if there's a edge from A to B, which we write as the pair (A, B), then A is an **in-neighbour** of B and B an **out-neighbour** of A. Node 4 has two out-neighbours and no in-neighbour.

The **degree** of a node is the number of edges attached to it, e.g. 'Alice' has degree 2. In a digraph, a node's **out-degree** is the number of outgoing edges and its **in-degree** is the number of incoming edges. Node 1 has out-degree 1 and in-degree 2; it therefore has degree 3.

Although the distinction between in- and out-neighbours and between in- and out-degrees only makes sense for digraphs, I'll use the terms for undirected graphs too, to simplify explanations.

In undirected graphs, edges (A, B) and (B, A) are the same, i.e. each edge can be seen as both outgoing and incoming, so the in- and the out-neighbours are the same as the neighbours and the in- and out-degree are the same as the degree. For example, 'Alice' and 'Bob' are both the in-neighbours and the out-neighbours of 'David'. Therefore the in-degree, out-degree and degree of 'David' are all 2.

In a digraph, the degree of a node is the sum of its in- and out- degrees. In an undirected graph, it's half the sum, because each edge counts both as incoming and outgoing.

What are the degrees of nodes 'Chan' and 3 in the above graphs?

Node 'Chan' has degree one and node 3 has degree zero.

Exercise 17.2.1

An advertising agency wants to advertise a product in some train stations. They give you the undirected graph of all the train stations in your country, with edges representing direct trains between them. The names of stations were replaced with numbers, to avoid bias by anyone working on the dataset. You're asked to produce a set of the train station(s) where the advert is likely to be seen by more people than in other stations.

Complete the following sentence:

I would select those nodes that ... because ...

Hint Answer

17.2.2 On graphs

A **path** from node *first* to node *last* is a sequence of distinct nodes

$(first, second, third, \dots, penultimate, last)$

such that edges $(first, second), (second, third), \dots, (penultimate, last)$ exist and are distinct. To put it bluntly, a path doesn't waste time going twice through a node or edge. The **length** of a path is the number of edges. Node B is **reachable** from node A if there's a path from A to B.

There may be multiple paths between the same nodes, e.g. (4, 1) and (4, 2, 1) are both paths from 4 to 1 in the example digraph. For most practical problems, we're interested in a **shortest path**, with the fewest edges.

In the undirected graph, sequence ('Alice', 'Bob', 'Alice') isn't a path because it repeats one node and one edge. Sequence ('Alice', 'David', 'Bob', 'Chan') is a path of length 3.

In the digraph, (1, 2, 1) isn't a path because, although it has no duplicate edges, it repeats node 1. Sequence (1, 4, 2) isn't a path either because edge (1, 4) doesn't exist.

A undirected graph is **connected** if there's one node, let's call it S, from which all other nodes are reachable; otherwise it's **disconnected**. It follows that in a connected undirected graph all nodes are mutually reachable. To see why, consider any two nodes A and B. There's a path from

S to A and one from S to B because the graph is connected. Since edges are undirected, the reverse paths from A to S and from B to S exist too. So, the paths from A to B via S and from B to A via S exist, proving that any nodes A and B are mutually reachable.

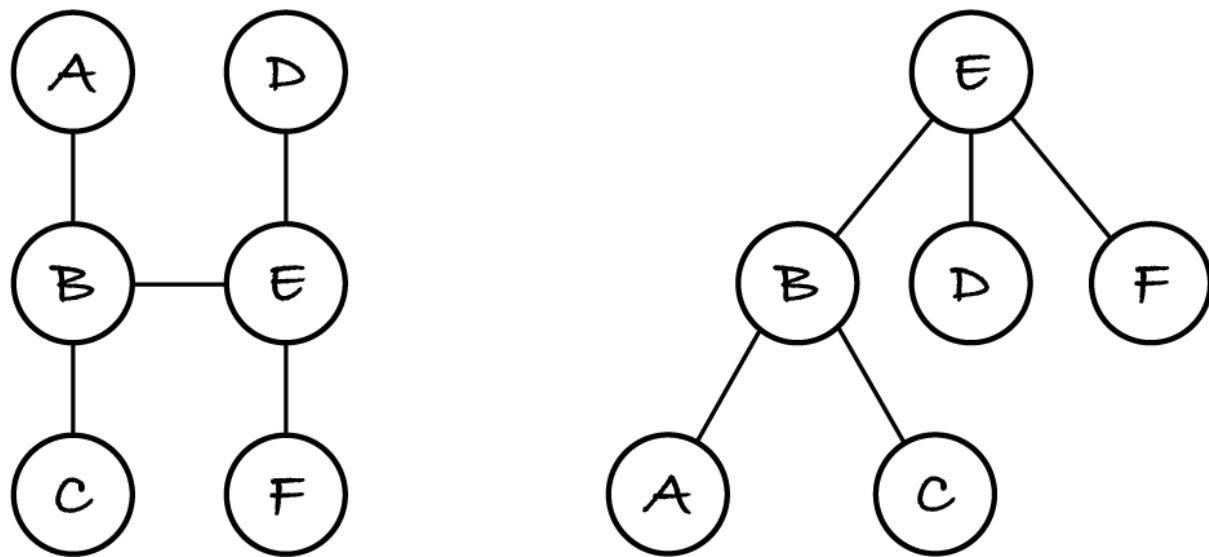
A digraph is connected (or disconnected) if, when removing the edge directions, we get a connected (or disconnected) undirected graph. The example digraph is disconnected because there's no path from or to node 3. We'll look at connectivity in digraphs *later*.

A **cycle** is a path except that the first and last nodes are the same. Since paths don't repeat edges, neither do cycles. Sequence ('Alice', 'Bob', 'Alice') isn't a cycle in the undirected graph, because it repeats one edge, but ('Alice', 'Bob', 'David', 'Alice') is. Sequence (1, 2, 1) is a cycle in the digraph, because it doesn't repeat edges.

A graph is **acyclic** if it has no cycles. A directed acyclic graph is abbreviated **DAG**.

The previous chapter introduced several kinds of rooted trees. We can now see them as special kinds of graphs. A **tree** is a connected acyclic undirected graph. A tree is *rooted* if one of its nodes is designated as the root. The neighbours of the root are the root's children, their neighbours are the nodes in level 2 of the rooted tree, and so on. The next figure shows on the left a tree and on the right the same tree rooted at node E.

Figure 17.2.2



In graphs, and therefore in trees, the neighbours of a node are in no particular order. In a rooted tree, one of the neighbours is the parent, but the other neighbours, the children, still aren't ordered. A *binary tree* is a rooted tree with ordered children: there's a left and a right child.

In a rooted tree there's a single path from any start node S to any other final node F : go up from S to the deepest common ancestor of S and F and then go down to F . For the tree rooted in E, the path from A to D goes through their deepest common ancestor, which happens to be the root: A, B, E, D. Since every tree can be transformed into a hierarchical tree by designating a root, it follows that in any tree there's a single path from any node to any other node. Another way to see that this must be so is to imagine there are two different paths from A to B. Then we could go from A to B following one path and return from B to A following the reverse of the other path. This would lead to a cycle, contrary to the assumption of being a tree.

A rooted tree with n nodes has $n - 1$ edges, because each node except the root has a single edge

to its parent. Since every tree can become a rooted tree, all trees have one edge less than the number of nodes.



Note: In M269, in the context of graphs, n refers to the number of nodes and e to the number of edges.

A graph is **dense** if it has many of the possible edges; it is **sparse** if it has few of the possible edges. The higher the ratio of edges to nodes, the denser the graph. Some graphs are clearly sparse, others are clearly dense, but for some it's in the eye of the beholder.

Trees are sparse because they have fewer edges than nodes, i.e. the edge/node ratio is less than one. Most large real-world networks have far more edges than nodes but nevertheless are clearly sparse because they have a tiny fraction of the possible edges. For example, in a city's road network, a junction typically only connects to three or four of the hundreds of junctions in the city, and in a social network we have very few friends among the millions of users.

17.2.3 Special graphs

The following graphs rarely occur when modelling real networks, but they're useful edge cases to test algorithms.

The **empty graph** has no nodes and hence no edges. A **null graph** is a non-empty graph without edges.

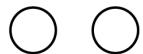
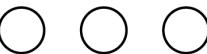
A **path graph** is a non-empty undirected graph in which the nodes can be numbered sequentially and the edges are between nodes 1 and 2, nodes 2 and 3, and so on. Path graphs are trees. If you extract a path from an undirected graph, you obtain a path graph.

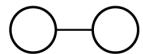
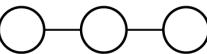
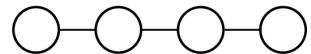
A **cycle graph** is an undirected graph with at least three nodes. It's obtained from a path graph by connecting the last node to the first. Cycle graphs are cyclic, hence their name.

A **complete graph** is an undirected graph where each node is connected to every other one.

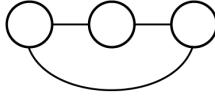
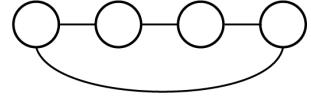
The next figure shows the null, path and complete graphs with one to four nodes, and the cycle graphs with three and four nodes. I haven't labelled the nodes.

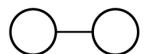
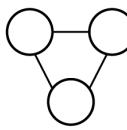
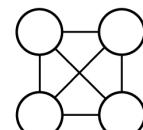
Figure 17.2.3

null graphs:    

path graphs:    

cycle graphs:

complete graphs:    

Note that some graphs are special in multiple ways, e.g. the graph with a single node is a null, path and complete graph, and the cycle graph with three nodes is also complete.

A cycle graph with one node would have an edge connecting the node to itself. A cycle graph with two nodes would have two edges between them. Neither situation is allowed by our undirected graph definition, so in M269 cycle graphs have at least three nodes.

Exercise 17.2.2

What are the degrees of the nodes in a

1. null
2. path
3. cycle
4. complete

graph with $n > 0$ nodes (and $n \geq 3$ for cycle graphs)?

Answer

Exercise 17.2.3

What is the number of edges in a

1. path
2. cycle
3. complete

graph with $n > 0$ nodes (and $n \geq 3$ for cycle graphs)? For each graph, write an expression based on n .

Answer

Exercise 17.2.4

Is a

1. null
2. path
3. cycle
4. complete

graph with many nodes sparse or dense?

Answer

17.2.4 ADT

Each of the above concepts can be turned into an operation for a graph ADT, e.g. an operation to check if two given nodes are mutually reachable, but the basic operations required are those on nodes and edges. Here are they for a digraph ADT:

Operation	Effect	In algorithms
new	create an empty graph	let g be an empty graph
has node	check if node a exists	a in g
add node	add a node a to a graph	add a to g
remove node	remove a node and its edges	remove a from g
has edge	check if an edge exists	(a, b) in g
add edge	add a directed edge	add (a, b) to g
remove edge	remove a directed edge	remove (a, b) from g
nodes	return the set of all nodes	nodes of g
edges	return the set of all directed edges	edges of g
in-neighbours	return a node's set of in-neighbours	in-neighbours of a in g
out-neighbours	return a node's set of out-neighbours	out-neighbours of a in g
neighbours	return the union of the in- and out-neighbours	neighbours of a in g
in-degree	return the in-degree of a node	in-degree of a in g
out-degree	return the out-degree of a node	out-degree of a in g
degree	return the sum of the in- and out-degree	degree of a in g

The undirected graph ADT has the same operations, but some have a different effect because they operate on undirected rather than directed edges:

- the operations on neighbours and degrees return the same set or the same number, for the same node
- the operations to add or remove an edge must add and remove the opposing edge too.

As an example of how graph algorithms will be written, here's a simple one, to isolate a given *node* in an undirected graph.

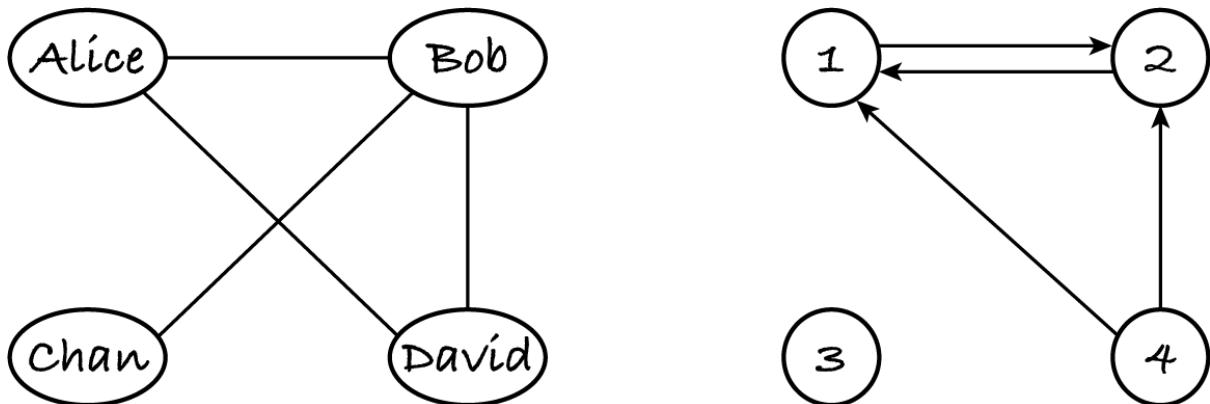
1. for each *neighbour* in neighbours of *node* in *graph*:
 1. remove edge (*node*, *neighbour*)

17.3 Edge list representation

The simplest way to store a graph is to have two separate collections, for its nodes and edges. This is called an **edge list** representation. The word ‘list’ doesn’t refer to Python’s list type, but rather to listing all edges explicitly. Any suitable data structure can be used for the collections: they’re usually arrays, linked lists or sets.

In Python, it’s best to use sets, one with the nodes, the other with pairs of nodes. Let’s consider the same example graphs.

Figure 17.3.1



The undirected graph on the left could be represented with sets

- `{'Alice', 'Bob', 'Chan', 'David'}`
- `{('Alice', 'Bob'), ('Bob', 'Chan'), ('Alice', 'David'), ('David', 'Bob')}`.

An undirected edge between nodes A and B is stored as a single pair to save memory: it doesn’t matter whether it’s (A, B) or (B, A). Since edges are undirected, one could consider representing each as a set of two nodes: `{{'Alice', 'Bob'}, {'Bob', 'Chan'}, ...}`. [Python’s sets](#) require items to be hashable, but sets themselves aren’t, so `{{'Alice', 'Bob'}, {'Bob', 'Chan'}}}` isn’t a valid Python literal.

The digraph on the right would be represented with sets

- `{1, 2, 3, 4}`
- `{(1, 2), (2, 1), (4, 1), (4, 2)}`.

Using Python’s `set` type has two advantages. The first is conceptual: the representation directly follows the definition of graphs as a set of nodes and a set of edges. The second is practical: the type supports constant-time membership checks, additions and deletions, whereas Python’s lists don’t.

If nodes can’t be represented by a hashable type (strings, integers, Booleans and tuples thereof), then Python’s lists are the second-best choice for an edge list representation.

17.3.1 Exercises

Exercise 17.3.1

Bob thinks that the collection of nodes helps implement graph operations in an easy and efficient way, but is not strictly necessary, since the nodes can be obtained by going through the collection of edges. He proposes to not store the collection of nodes for very large graphs, to save memory.

Is he right? Is the collection of nodes redundant? Can we implement the graph ADT just with the collection of edges?

Hint Answer

Exercise 17.3.2

For each operation below, describe very briefly how it's implemented for digraphs and give the worst-case complexity in terms of the number of nodes n and the number of edges e . You may wish to look again at the complexity of *set operations*. I've done some operations for you.

Operation	Implementation	Complexity
add node A	add to set of nodes	$\Theta(1)$
remove node A		
has edge (A, B)	check membership in the set of edges	$\Theta(1)$
add edge (A, B)		
remove edge (A, B)		
in-neighbours of A	find all B such that (B, A) is in the set of edges	$\Theta(e)$
out-neighbours of A		
in-degree of A		

Hint Answer

Exercise 17.3.3

Our definition of graph doesn't allow **loops** (edges between the same node) but sometimes they are necessary to model a network. It's therefore useful to know how to handle them.

Consider a director/actor network. It can be modelled with a digraph where the nodes represent people and where edge (A, B) means that A has directed B in some play or film. If person A acted in a play or film they directed, then the digraph must include edge (A, A).

Can an edge list representation accommodate loops? How?

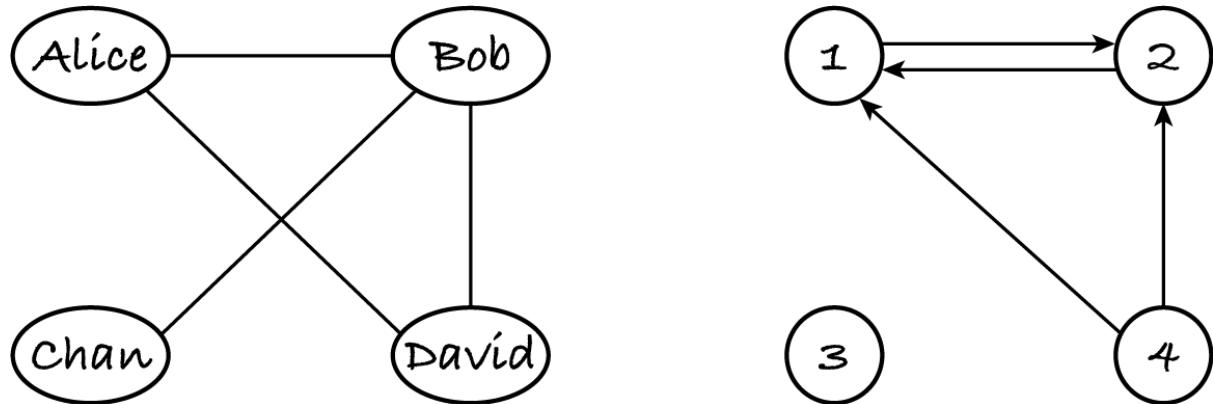
Answer

17.4 Adjacency matrix representation

A graph with n nodes can be stored as an **adjacency matrix** of $n \times n$ Booleans: the Boolean in row A and column B is true if and only if there's an edge from A to B. In other words, each row of the matrix indicates the out-neighbours of that row's node.

Consider the same example graphs.

Figure 17.4.1



Adjacency matrices use node numbers as indices. Unless the graph happens to have nodes labelled 0, 1, 2, ..., we must number the nodes and keep a map from natural numbers to nodes, in addition to the matrix. The map can be implemented as a lookup table, a hash table or a BST. Here's one possible representation of the undirected graph in Python, with a lookup table in the form of a list.

```
nodes = ['Alice', 'Bob', 'Chan', 'David']
edges = [
    [False, True, False, True],    # neighbours of node 0 ('Alice'):_
    ↪nodes 1 and 3
    [True, False, True, True],    # neighbours of node 1 ('Bob'):_
    ↪nodes 0, 2, 3
    [False, True, False, False],   # neighbours of node 2 ('Chan'):_
    ↪node 1
    [True, True, False, False]    # neighbours of node 3 ('David'):_
    ↪nodes 0 and 1
]
```

For example, Alice's neighbours are Bob and David, which are nodes numbered 1 and 3. So the row for Alice, row 0, has positions 1 and 3 set to true and the others set to false.

The digraph could be represented in Python as follows.

```
nodes = [1, 2, 3, 4]
edges = [
    [False, True, False, False],   # out-neighbours of node 0_
    ↪(labelled 1)
    [True, False, False, False],   # out-neighbours of node 1_
    ↪(labelled 2)
    [False, False, False, False],  # out-neighbours of node 2_
    ↪(labelled 3): none
    [True, True, False, False]    # out-neighbours of node 3_
    ↪(labelled 4)
]
```

Adjacency matrices are usually only used for small graphs with a fixed set of nodes, for the following reasons.

Adjacency matrices for undirected graphs are mirrored along the main diagonal, from the top left to the bottom right. Row A, column B and row B, column A are either both true or both false: either there is an undirected edge between A and B or there isn't. Half of the matrix is wasted memory as it can be inferred from the other half.

An adjacency matrix includes all *potential* edges; an edge list only includes the *actual* edges. If a graph is sparse, then an edge list data structure is likely to occupy far less memory than $n \times n$ Booleans. If a graph is dense, then an edge list structure (one pair per edge) is likely to occupy more memory than the adjacency matrix (one Boolean per edge). Since most real-world networks are sparse, adjacency matrices are rarely used in practice.

Matrices make adding and removing nodes more complicated. To add a node, we put it in the nodes map and append a row and a column to the matrix. To remove a node, we remove it from the nodes map and remove the corresponding row and column from the matrix, which is inefficient if the matrix is stored as an array of arrays.

17.4.1 Exercises

Exercise 17.4.1

Briefly describe the following operations on digraphs and their complexity in terms of n and e , the number of nodes and edges, when using an adjacency matrix stored as an array of arrays. For this exercise, you can ignore the additional node map.

Operation	Implementation	Complexity
add node A	append a row and a column	$\Theta(n)$
remove node A	remove A's row and column	$\Theta(n^2)$
has edge (A, B)		
add edge (A, B)		
remove edge (A, B)		
in-neighbours of A		
out-neighbours of A		
in-degree of A		

Answer

Exercise 17.4.2

Comparing the complexity of the operations in your table above to their *complexity for sets of edges*, which representation is usually more efficient, assuming nodes are infrequently added?

Hint Answer

Exercise 17.4.3

Can an adjacency matrix represent loops? How?

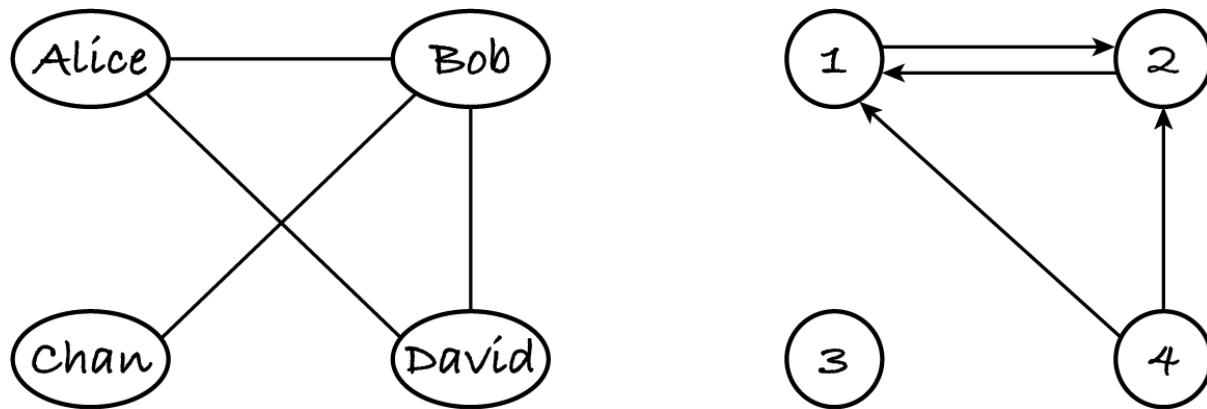
Answer

17.5 Adjacency list representation

The row for node A in the adjacency matrix indicates, with true values, the out-neighbours of A. A more compact representation is to only list the out-neighbours and to omit all other nodes.

Let's consider the same example graphs.

Figure 17.5.1



The example undirected graph is represented as follows.

```
nodes = ['Alice', 'Bob', 'Chan', 'David']
edges = [
    [1, 3],      # neighbours of node 0 ('Alice')
    [0, 2, 3],   # neighbours of node 1 ('Bob')
    [1],         # neighbours of node 2 ('Chan')
    [0, 1]       # neighbours of node 3 ('David')
]
```

This contains the same information as the adjacency matrix: each row of n Booleans is represented as the indices of the true values.

Instead of an adjacency matrix we have a lookup table of sequences. Since most real-world networks are sparse, each sequence is usually very short, compared to the number of nodes. The length of the sequence for node A is the out-degree of A.

For an undirected graph, each edge (A, B) is stored twice: A is in the neighbours of B and B is in the neighbours of A.

The digraph can be represented like this:

```
nodes = [1, 2, 3, 4]
edges = [
    [1],        # out-neighbours of node 0 (labelled 1)
```

(continues on next page)

(continued from previous page)

```
[0],          # out-neighbours of node 1 (labelled 2)
[],           # out-neighbours of node 2 (labelled 3)
[0, 1]        # out-neighbours of node 3 (labelled 4)
]
```

If nodes are of a hashable type, we can use a single hash table to store the nodes *and* their out-neighbours. The digraph would be stored in dictionary `{ 1: [2], 2: [1], 3: [], 4: [1, 2] }` and the undirected graph in dictionary

```
{
    'Alice': ['Bob', 'David'],
    'Bob': ['Alice', 'Chan', 'David'],
    'Chan': ['Bob'],
    'David': ['Alice', 'Bob']
}
```

There are further variants of this representation, like using linked lists instead of arrays or also storing the in-neighbours of each node. An **adjacency list** representation of a graph is a map of nodes to their in- or out-neighbours. Each entry in the map, i.e. each collection of in- or out-neighbours, is called that node's adjacency list. Again, the word 'list' is used in the general sense of listing: the adjacency lists can be of any suitable collection type.

Let's assume that each adjacency list only includes the out-neighbours. This means that every directed edge (A, B) is represented once, namely by having out-neighbour B in the adjacency list for A. If the graph is undirected, then each edge is stored twice as we've seen above.

The consequence of this is that any graph operation that goes through all the out-neighbours of all the nodes is effectively going through all the nodes once and through all the edges once or twice and thus has complexity $\Theta(n) + \Theta(e)$. Realistic graphs tend to have more edges than nodes, so we simplify to $\Theta(e)$. This eases the complexity analysis of graph algorithms.



Note: An algorithm that goes through each out-neighbour of each node is going through all the edges. This takes $\Theta(e)$ time for adjacency lists of out-neighbours.

17.5.1 Exercises

Exercise 17.5.1

For each operation below, describe briefly its implementation for a digraph and state its complexity. Assume that the graph is stored as a Python dictionary of nodes to lists of out-neighbours, as shown above. Remember that all operations on dictionaries take constant time.

Operation	Implementation	Complexity
add node A		
remove node A	remove A from all lists and delete the entry for A	$\Theta(e)$
has edge (A, B)	check if B is in the list for A	$\Theta(\text{out-degree}(A))$
add edge (A, B)		
remove edge (A, B)		
in-neighbours of A		
out-neighbours of A		
in-degree of A		

Answer

Exercise 17.5.2

We can represent each adjacency list with a Python set instead of a Python list. The example digraph is stored as `{ 1: {2}, 2: {1}, 3: set(), 4: {1, 2} }`.

State the complexities for the operations on a digraph, for this representation.

Operation	Complexity
add node A	
remove node A	
has edge (A, B)	
add edge (A, B)	
remove edge (A, B)	
in-neighbours of A	
out-neighbours of A	
in-degree of A	

Hint Answer

Exercise 17.5.3

Can an adjacency list representation accommodate loops? How?

Answer

17.6 Classes for graphs

Looking at the worst-case complexities of the operations for the various graph data structures, using a map of sets of out-neighbours is a clear winner: all operations (except returning all edges) take at most linear time in the number of nodes. For some representations, the same operation is linear in the number of edges. This takes longer for most graphs, which have more edges than nodes.



Note: A graph algorithm with a complexity based on n , e.g. $\Theta(n^2)$, is preferable to an algorithm with the same complexity based on e , e.g. $\Theta(e^2)$, because $n < e$ for most graphs.

Having decided the ADT's operations and how to represent graphs, I proceed to define two Python classes, for digraphs and for undirected graphs.

17.6.1 The `DiGraph` class

My implementation of a digraph with an adjacency list representation is below. Nodes must be represented by hashable objects due to the use of a dictionary. Class `Hashable` in the `typing` module allows us to indicate which function parameters must be hashable.

I've added a method to draw graphs, so that we can visually check them. The method uses the `NetworkX` module, which is included in the M269 software but isn't part of Python's standard library.

Make sure you understand all the code, except the draw method. You may wish to remind yourself of Python's *dictionary and set operations*. If anything's unclear, ask in the M269 forums or tutorials.

```
[1]: # this code is also in m269_digraph.py

import networkx
from typing import Hashable

class DiGraph:
    """A directed graph with hashable node objects.

    Edges are between different nodes.
    There's at most one edge from one node to another.
    """

    def __init__(self) -> None:
        """Create an empty graph."""
        self.out = dict() # a map of nodes to their out-neighbours

    def has_node(self, node: Hashable) -> bool:
        """Return True if and only if the graph has the node."""
        return node in self.out

    def has_edge(self, start: Hashable, end: Hashable) -> bool:
        """Return True if and only if edge start -> end exists.

        Preconditions: self.has_node(start) and self.has_node(end)
        """

```

(continues on next page)

(continued from previous page)

```

        return end in self.out[start]

def add_node(self, node: Hashable) -> None:
    """Add the node to the graph.

    Preconditions: not self.has_node(node)
    """
    self.out[node] = set()

def add_edge(self, start: Hashable, end: Hashable) -> None:
    """Add edge start -> end to the graph.

    If the edge already exists, do nothing.

    Preconditions:
    self.has_node(start) and self.has_node(end) and start != end
    """
    self.out[start].add(end)

def remove_node(self, node: Hashable) -> None:
    """Remove the node and all its attached edges.

    Preconditions: self.has_node(node)
    """
    for start in self.out:
        self.remove_edge(start, node)
    self.out.pop(node)

def remove_edge(self, start: Hashable, end: Hashable) -> None:
    """Remove edge start -> end from the graph.

    If the edge doesn't exist, do nothing.

    Preconditions: self.has_node(start) and self.has_node(end)
    """
    self.out[start].discard(end)

def nodes(self) -> set:
    """Return the graph's nodes."""
    all_nodes = set()
    for node in self.out:
        all_nodes.add(node)
    return all_nodes

def edges(self) -> set:
    """Return the graph's edges as a set of pairs (start, end)."""

```

(continues on next page)

(continued from previous page)

```

→"
    all_edges = set()
    for start in self.out:
        for end in self.out[start]:
            all_edges.add((start, end))
    return all_edges

def out_neighbours(self, node: Hashable) -> set:
    """Return the out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return set(self.out[node]) # return a copy

def out_degree(self, node: Hashable) -> int:
    """Return the number of out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return len(self.out[node])

def in_neighbours(self, node: Hashable) -> set:
    """Return the in-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    start_nodes = set()
    for start in self.out:
        if self.has_edge(start, node):
            start_nodes.add(start)
    return start_nodes

def in_degree(self, node: Hashable) -> int:
    """Return the number of in-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return len(self.in_neighbours(node))

def neighbours(self, node: Hashable) -> set:
    """Return the in- and out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return self.out_neighbours(node).union(self.in_
→neighbours(node))

```

(continues on next page)

(continued from previous page)

```

def degree(self, node: Hashable) -> int:
    """Return the number of in- and out-going edges of the node.

    Preconditions: self.has_node(node)
    """
    return self.in_degree(node) + self.out_degree(node)

def draw(self) -> None:
    """Draw the graph."""
    if type(self) is DiGraph:
        graph = networkx.DiGraph()
    else:
        graph = networkx.Graph()
    graph.add_nodes_from(self.nodes())
    graph.add_edges_from(self.edges())
    networkx.draw(
        graph,
        with_labels=True,
        node_size=1000,
        node_color="lightblue",
        font_size=12,
        font_weight="bold",
    )
    )

```

Exercise 17.6.1 (optional)

Here are some prompts to discuss your understanding of the code with others:

1. Why does the `out-neighbours` method waste memory and return a copy?
2. The in- and out-degree are computed as the number of in- and out-neighbours. Why doesn't the `degree` method just return the number of neighbours?
3. The `in_degree` method can be changed to run faster and use less memory. How?
4. Why did I implement `remove_edge` so that it works for non-existing edges?

17.6.2 The UndirectedGraph class

Since an undirected graph can be represented as a digraph with opposing edges between every pair of nodes, I implement a subclass of `DiGraph` to reuse the representation and methods.

I must redefine the methods that add and remove edges, so that they also add and remove the opposite edge. Python's `super()` function allows us to access the methods of the superclass. I must also reimplement other operations, to fit the *definitions for undirected graphs*.

[2]: # this code is also in m269_ungraph.py

(continues on next page)

(continued from previous page)

```

from typing import Hashable

class UndirectedGraph(DiGraph):
    """An undirected graph with hashable node objects.

    There's at most one edge between two different nodes.
    There are no edges between a node and itself.
    """

    def add_edge(self, node1: Hashable, node2: Hashable) -> None:
        """Add an undirected edge node1-node2 to the graph.

        If the edge already exists, do nothing.

        Preconditions: self.has_node(node1) and self.has_node(node2)
        """
        super().add_edge(node1, node2)
        super().add_edge(node2, node1)

    def remove_edge(self, node1: Hashable, node2: Hashable) -> None:
        """Remove edge node1-node2 from the graph.

        If the edge doesn't exist, do nothing.

        Preconditions: self.has_node(node1) and self.has_node(node2)
        """
        super().remove_edge(node1, node2)
        super().remove_edge(node2, node1)

    def edges(self) -> set:
        """Return the graph's edges as a set of pairs.

        Postconditions: for every edge A-B,
        the output has either (A, B) or (B, A) but not both
        """
        all_edges = set()
        for node1 in self.out:
            for node2 in self.out[node1]:
                if (node2, node1) not in all_edges:
                    all_edges.add((node1, node2))
        return all_edges

    def in_neighbours(self, node: Hashable) -> set:
        """Return all nodes that are adjacent to the node.

```

(continues on next page)

(continued from previous page)

```

Preconditions: self.has_node(node)
"""

return self.out_neighbours(node)

def neighbours(self, node: Hashable) -> set:
    """Return all nodes that are adjacent to the node.

Preconditions: self.has_node(node)
"""

return self.out_neighbours(node)

def in_degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

Preconditions: self.has_node(node)
"""

return self.out_degree(node)

def degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

Preconditions: self.has_node(node)
"""

return self.out_degree(node)

```

Exercise 17.6.2 (optional)

You may wish to discuss the following with your study buddy or peers:

1. If the `in_neighbours` and `neighbours` methods had been inherited instead of reimplemented, would they work for undirected graphs? If so, why did I reimplement them?
2. Same questions, but for the `in_degree` and `degree` methods.

17.6.3 Special graphs

We can now construct the *special graphs* introduced earlier.

```
[3]: # this code is also in m269_graphs.py

EMPTY_UG = UndirectedGraph()

def null_graph(n: int) -> UndirectedGraph:
    """Return a graph with nodes 0, 1, ..., n-1 and no edges.
```

(continues on next page)

(continued from previous page)

```
Preconditions: n > 0
"""
graph = UndirectedGraph()
for node in range(n):
    graph.add_node(node)
return graph

def path_graph(n: int) -> UndirectedGraph:
    """Return a graph with nodes 0, ..., n-1 and edges 0-1, 1-2, ..."""

    Preconditions: n > 0
    """
    graph = null_graph(n)
    for node in range(n - 1):
        graph.add_edge(node, node + 1)
    return graph

def cycle_graph(n: int) -> UndirectedGraph:
    """Return a graph with nodes 0, ..., n-1 and edges 0-1, 1-2, ..., (n-1)-0.

    Preconditions: n > 0
    """
    graph = path_graph(n)
    graph.add_edge(n - 1, 0)
    return graph

def complete_graph(n: int) -> UndirectedGraph:
    """Return a graph with nodes 0, ..., n-1 connected to each other.

    Preconditions: n > 0
    """
    graph = null_graph(n)
    for node1 in range(n):
        for node2 in range(node1 + 1, n):
            graph.add_edge(node1, node2)
    return graph
```

NetworkX draws graphs with the same `matplotlib` module I've used earlier to *process images*. We must tell Jupyter to draw the graphs in the notebook.

```
[4]: %matplotlib inline
```

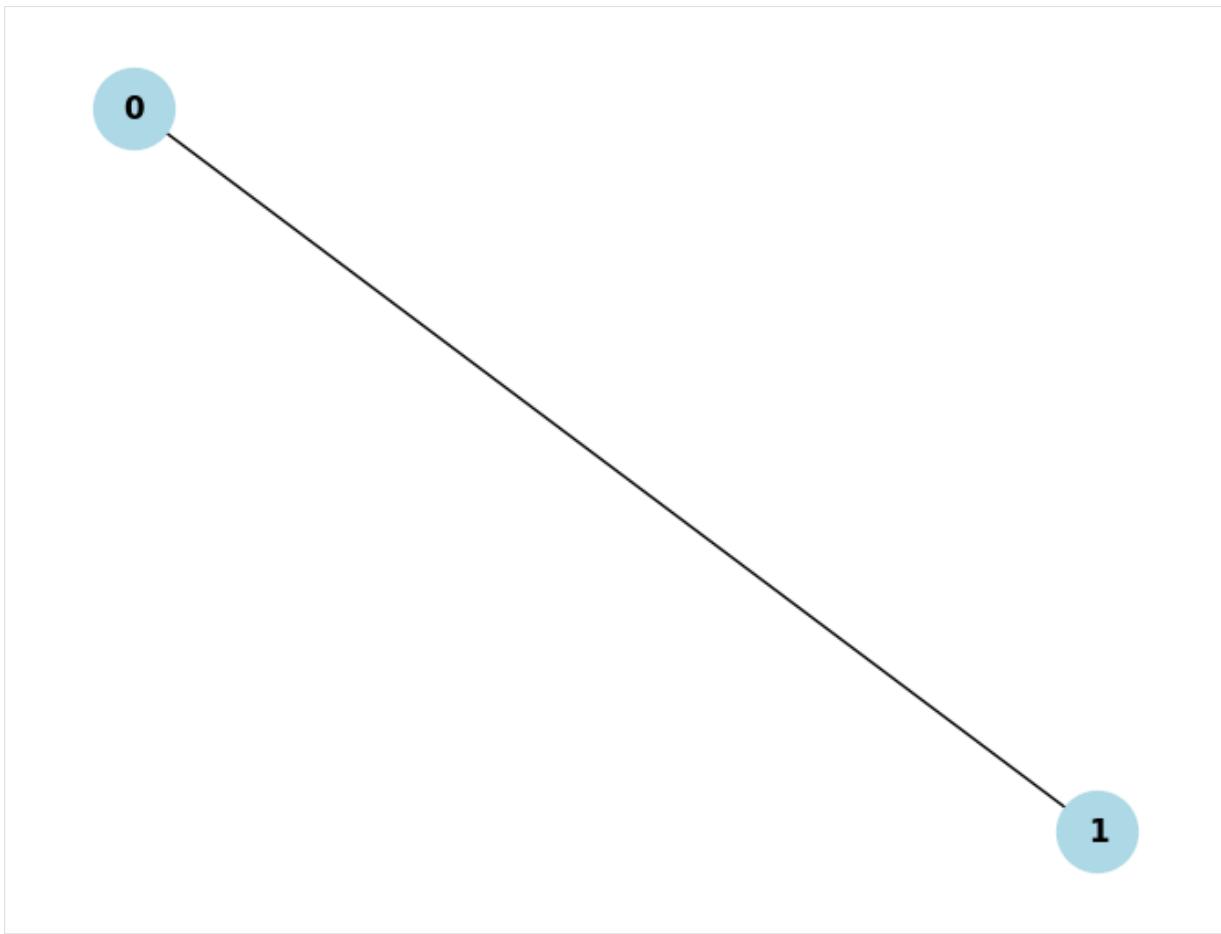
If a code cell draws several graphs, only the last one is shown, so we must draw one graph per

cell.

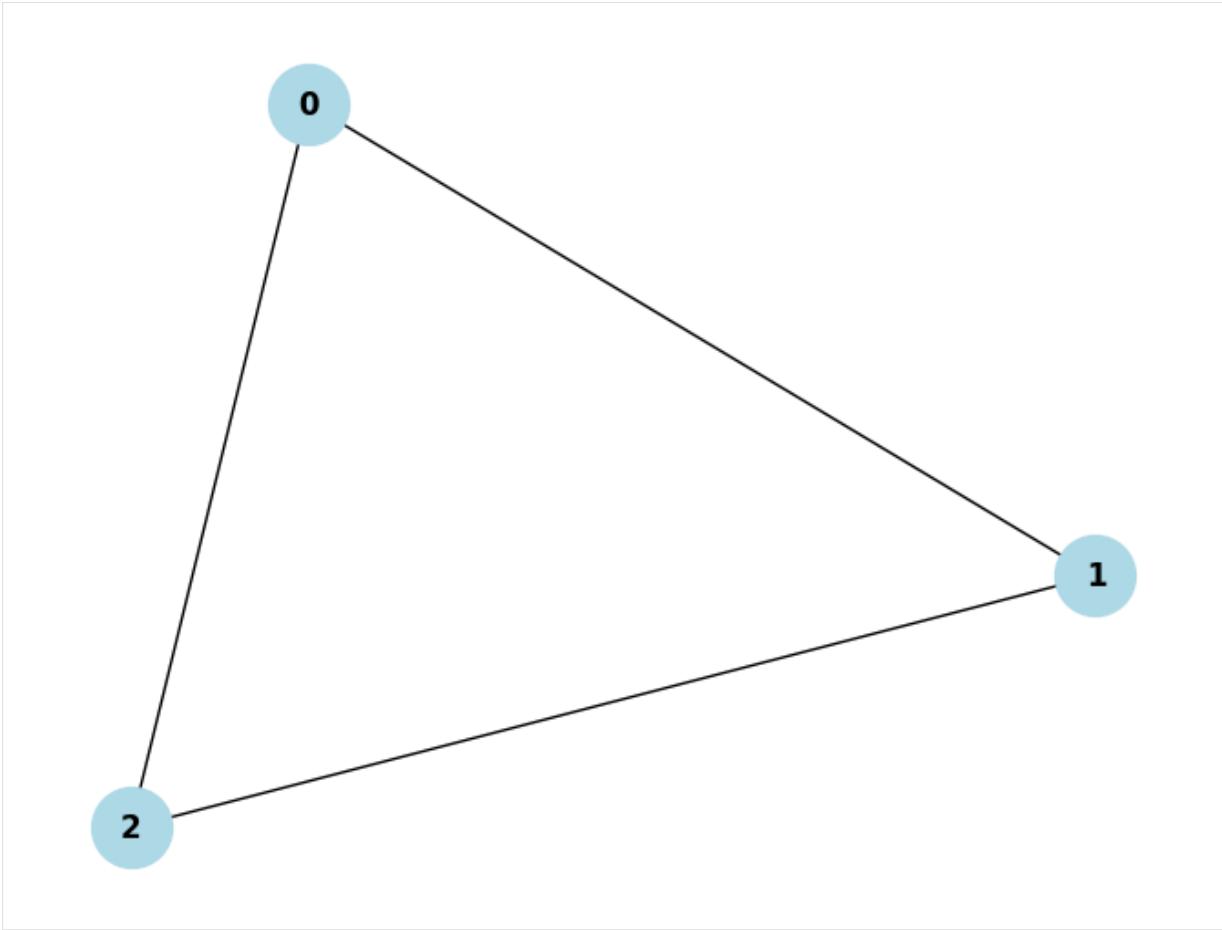
```
[5]: null_graph(1).draw()
```



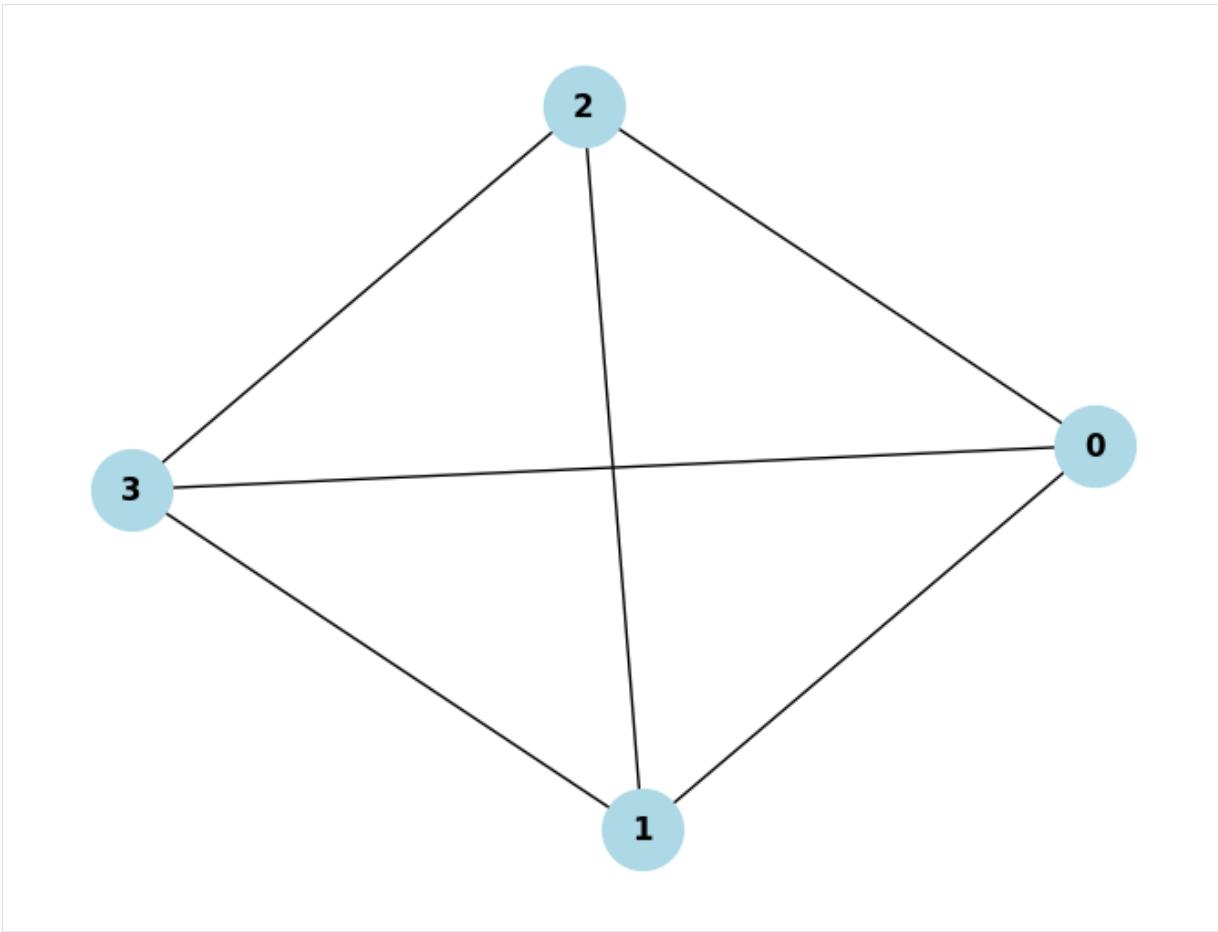
```
[6]: path_graph(2).draw()
```



```
[7]: cycle_graph(3).draw()
```



```
[8]: complete_graph(4).draw()
```



A graph's layout is computed from an initially random position of the nodes. Every time you run a cell that draws a graph, the layout may change.

Exercise 17.6.3

Complete the function below.

```
[9]: from algoesup import test

def same_degree(graph: UndirectedGraph) -> bool:
    """Return True if and only if all nodes have the same degree.

    Preconditions: graph isn't empty
    """
    pass

same_degree_tests = [
    #case,           graph,           same degree?
    ('null',        null_graph(1),   True),   # all have degree 0
    ('path 2',      path_graph(2),   True),   # all have degree 1
    ('path 3',      path_graph(3),   False),
]
```

(continues on next page)

(continued from previous page)

```

('cycle',      cycle_graph(3),      True),  # all have degree 2
('complete',   complete_graph(4),   True)   # all have degree 3
]

test(same_degree, same_degree_tests)

```

Hint Answer

17.6.4 Random graphs

To test graph algorithms, it's useful to have less 'predictable' graphs. We can generate a random graph by including each edge with some probability p . (Remember that a probability is given by a real number from 0 to 1.)

The higher the p chosen, the denser the resulting graph is, because of the larger likelihood for edges to be included. What graphs are generated for $p = 0$ and for $p = 1$?

Those values of p generate a null graph and a complete graph, respectively.

Let's implement the approach, using function `random` in module `random` to generate a float from 0 (inclusive) to 1 (exclusive).

[10]: # this code is also in m269_graphs.py

```

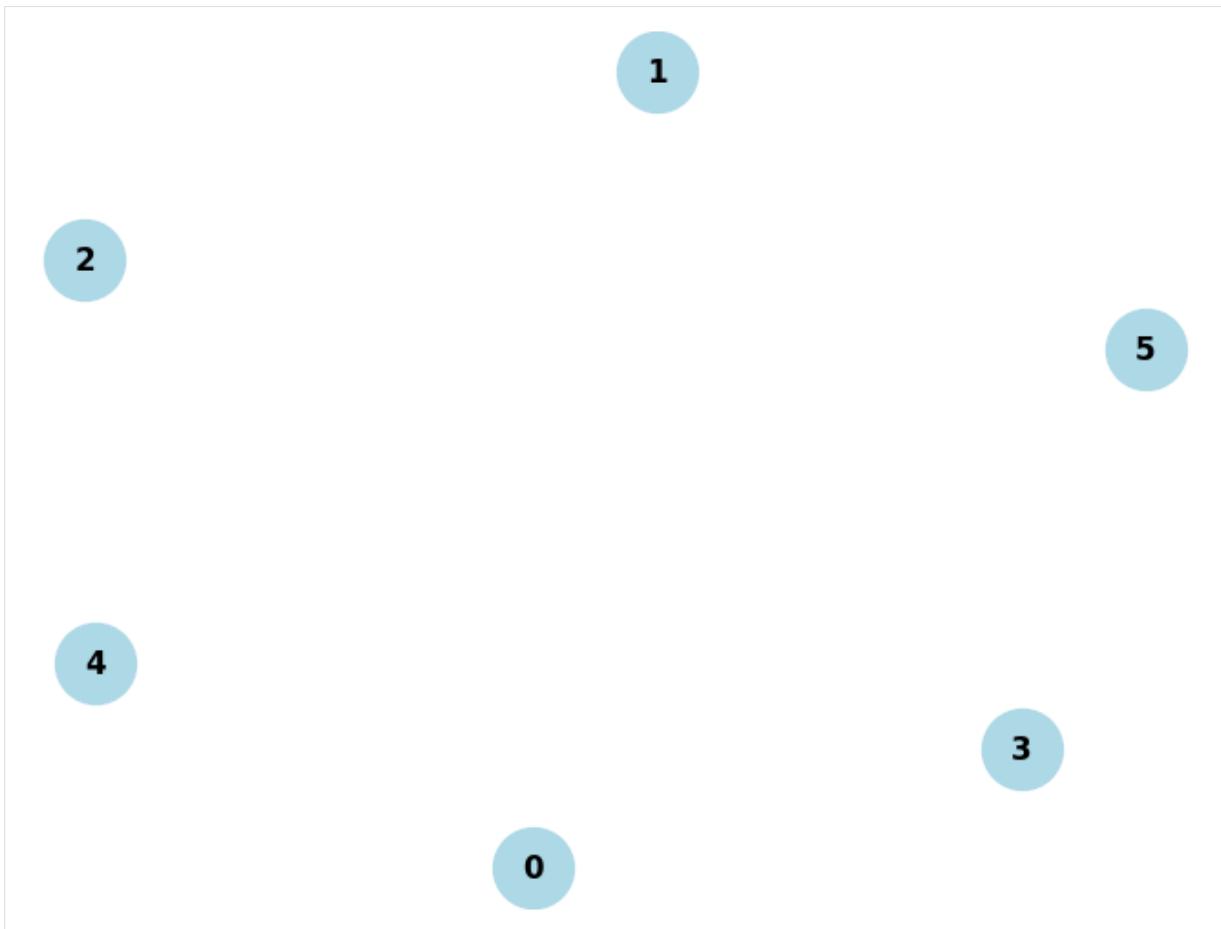
from random import random

def random_graph(n: int, probability: float) -> UndirectedGraph:
    """Generate a random graph with n nodes.

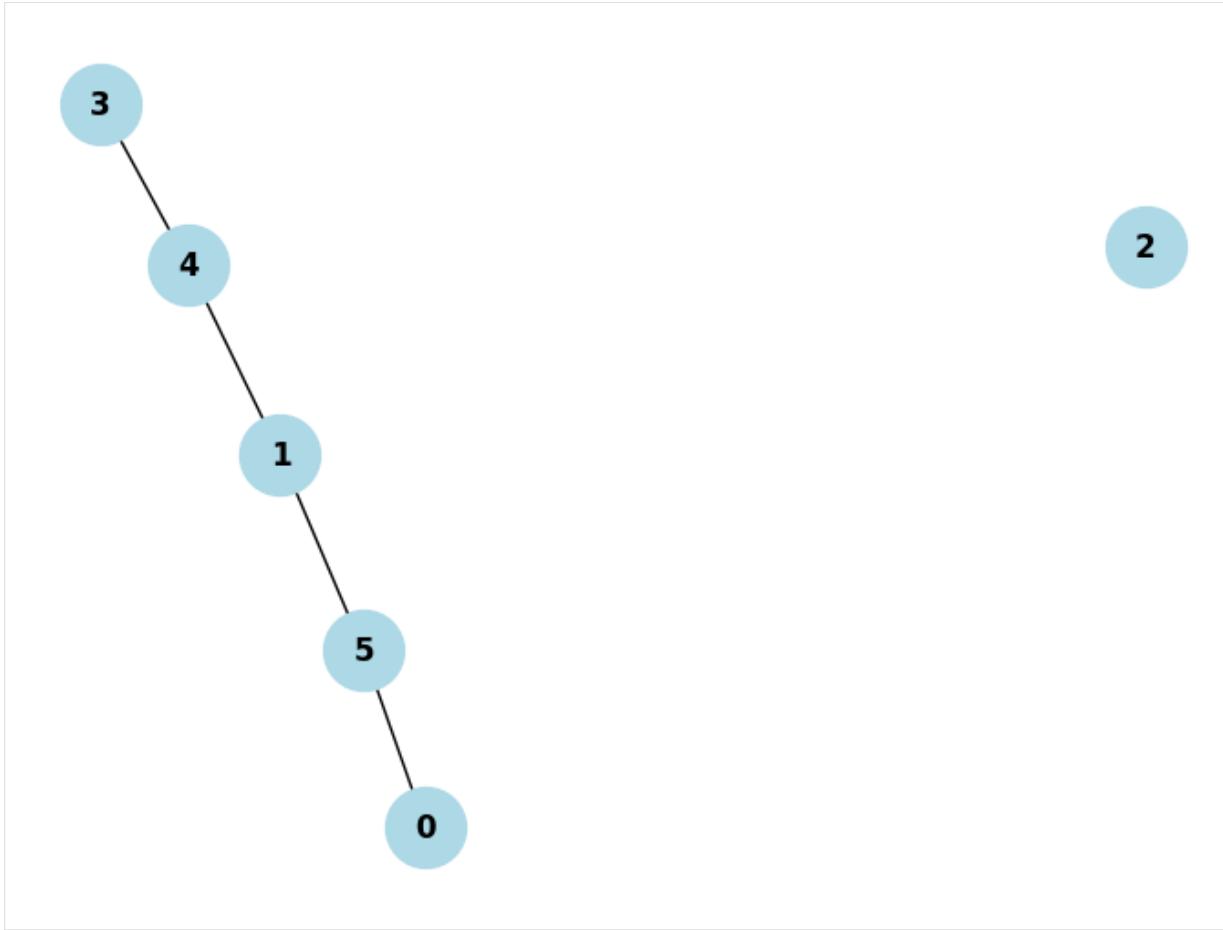
    Each edge has the given probability of existing.
    Preconditions: 0 <= probability <= 1
    """
    graph = null_graph(n)
    for node1 in range(n):
        for node2 in range(node1 + 1, n):
            if random() < probability:
                graph.add_edge(node1, node2)
    return graph

```

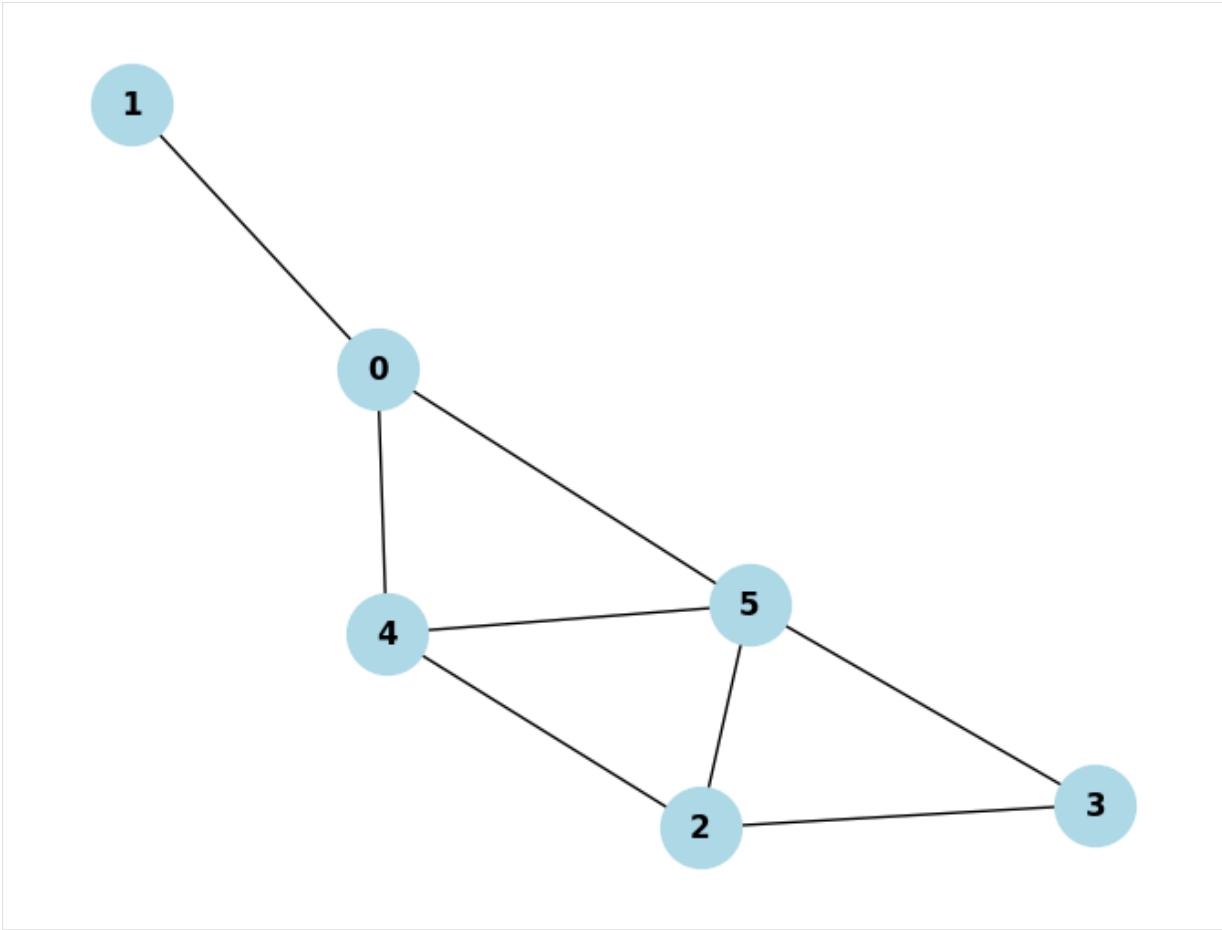
[11]: random_graph(6, 0).draw() # p = 0%: no edges



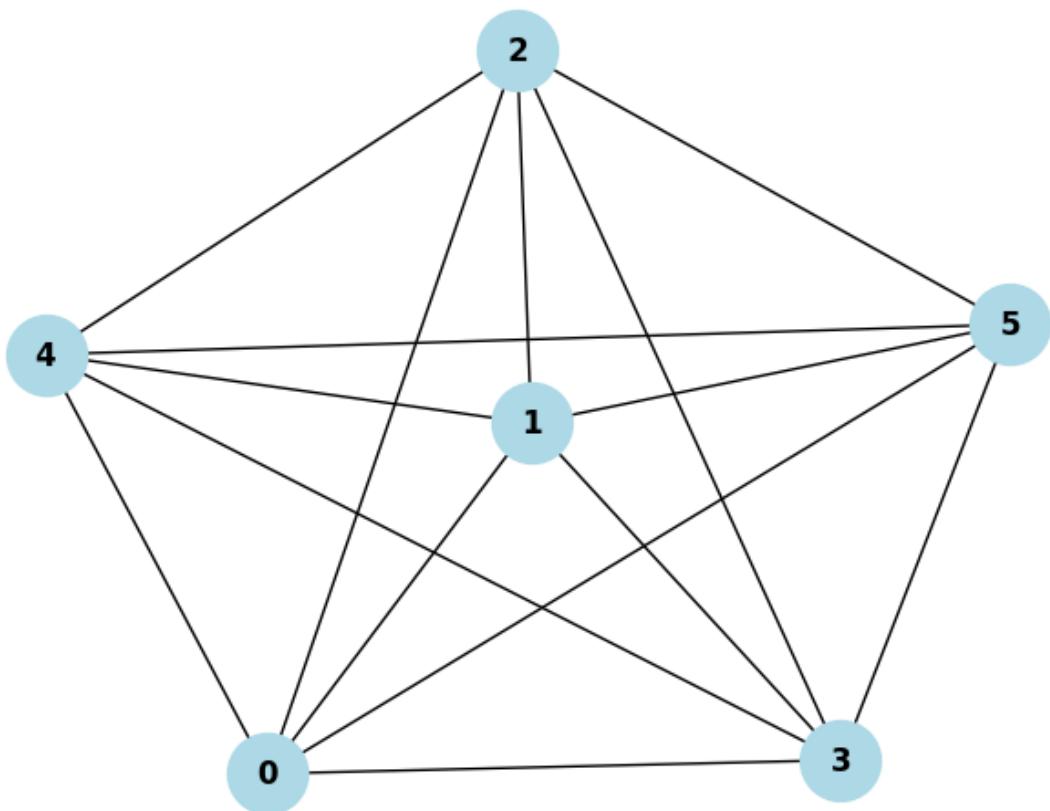
```
[12]: random_graph(6, 0.4).draw() # p = 40%
```



```
[13]: random_graph(6, 0.4).draw() # different graph for same n and p
```



```
[14]: random_graph(6, 1).draw() # p = 100%: all edges
```



Info: Edgar Gilbert proposed this random graph generation approach in 1959.

17.7 Traversing a graph

Several algorithms on rooted trees, e.g. to compute a tree's height or to evaluate an infix expression represented as a tree, are based on *traversing the tree*, either in breadth or in depth. Likewise, several graph algorithms are based on traversing the graph, visiting each node once. A graph has no designated root node, so a traversal algorithm takes as input the graph and a start node.

17.7.1 First algorithm

We can traverse a graph by following the outgoing edges of previously visited nodes in order to visit further nodes. We can only visit those nodes reachable from the start node. In a graph there may be multiple paths from the start node to any other node. To avoid visiting a node twice, we track which nodes we have visited.

We keep two sets of nodes: those already visited and those still unprocessed. Initially, the start node has been visited and its out-neighbours are unprocessed. While there are nodes to be

processed, we pick one of them and check if it was already visited. If not, we visit the node and add its out-neighbours to the unprocessed nodes.

Here's the algorithm, with inputs *graph* and *start*, which is a node of *graph*.

1. let *visited* be $\{start\}$
2. let *unprocessed* be the out-neighbours of *start* in *graph*
3. while *unprocessed* isn't empty:
 1. remove one *node* from *unprocessed*
 2. if *node* not in *visited*:
 1. visit *node*
 2. add *node* to *visited*
 3. for each *neighbour* in out-neighbours of *node* in *graph*:
 1. add *neighbour* to *unprocessed*

What we do when visiting a node (step 3.2.1) depends on the problem at hand.

Step 3.2 ensures two things: each node is visited at most once and only the out-neighbours of unvisited nodes are marked for processing later. Eventually the traversal has visited all nodes it can reach from the start node. From then on, the loop keeps processing nodes, but they've all been visited and so no further nodes are added for processing. At some point, the set of unprocessed nodes becomes empty and the algorithm stops.

17.7.2 Complexity

Let's analyse the complexity of the traversal itself, ignoring step 3.2.1.

What's a best-case scenario and the corresponding complexity?

The algorithm does the least work if it only visits the start node and doesn't enter the loop, i.e. *unprocessed* is empty. This happens if the start node has no neighbours. In such a best-case scenario, steps 1 and 2 take constant time.

The algorithm does the most work when all nodes are reachable from the start node. In a worst-case scenario, it visits all n nodes.

It's sometimes easier to analyse an algorithm if we look at the overall complexity of each non-loop statement, instead of looking at the complexity of a single execution and then multiplying by the number of iterations.

Here's the algorithm again, annotated with the total worst-case complexity of each step, except for loops and for visiting nodes.

1. $\Theta(1)$ let *visited* be $\{start\}$
2. $\Theta(\text{out-degree}(start))$ let *unprocessed* be the out-neighbours of *start* in *graph*
3. while *unprocessed* isn't empty:

1. $\Theta(e)$ remove one *node* from *unprocessed*
2. $\Theta(e)$ if *node* not in *visited*:
 1. visit *node*
 2. $\Theta(n)$ add *node* to *visited*
 3. for each *neighbour* in out-neighbours of *node* in *graph*:
 1. $\Theta(e)$ add *neighbour* to *unprocessed*

The initialisation assignments are done only once.

Because of step 3.2, each node is visited once, so step 3.2.2 is executed n times in total.

Step 3.2.3 goes through every out-neighbour of each visited node and all nodes are visited in a worst-case scenario, so the algorithm goes through all edges. In other words, every edge is added to set *unprocessed* in step 3.2.3.1 and later removed in step 3.1. The while-loop is executed e times and so is step 3.2.

All operations on sets take constant time, so the worst-case complexity is

$$\Theta(1) + \Theta(\text{out-degree}(\text{start})) + \Theta(n) + 3 \times \Theta(e) = \Theta(n + e).$$

As usual, we ignore constant factors and the faster operations: the out-degree of *start* is never more than the total number of edges, so $\Theta(\text{out-degree}(\text{start})) + \Theta(n) = \Theta(n)$.

Most graphs have more edges than nodes, so $\Theta(n + e) = \Theta(e)$ for them. But some graphs have more nodes than edges, so $\Theta(n + e) = \Theta(n)$ for them. We could therefore say that the worst-case complexity of traversing a graph is $\Theta(n)$ if it has more nodes than edges, otherwise it is $\Theta(e)$.

The densest graphs have $n \times (n - 1)$ edges, so $\Theta(n + e) = \Theta(n) + O(n^2) = O(n^2)$ for any graph. Once we replace a graph's exact number of edges e with an upper bound (the maximal number of edges), we must use *Big-Oh* instead of Big-Theta notation.

In summary, we can give the complexity of graph algorithms in terms of just the number of edges or just the number of nodes, but it's simpler, more general and more precise to state it in terms of both.



Note: State the complexity of graph algorithms in terms of the number of nodes n and the number of edges e .

17.7.3 Code and tests

Let's implement and test the algorithm. I must first load both graph classes and the special graphs.

```
[1]: %run -i ../m269_digraph
%run -i ../m269_ungraph
%run -i ../m269_graphs
```

Instead of a set, I keep the visited nodes in a sequence and return it, to see in which order the nodes were visited.

```
[2]: def traversal(graph: DiGraph, start: Hashable) -> list:
    """Return all nodes reachable from start, in the order visited.

    Preconditions: graph.has_node(start)
    """

    visited = [start]
    unprocessed = graph.out_neighbours(start)
    while len(unprocessed) > 0:
        node = unprocessed.pop()
        if node not in visited:
            visited.append(node)
            for neighbour in graph.out_neighbours(node):
                unprocessed.add(neighbour)
    return visited
```

```
[3]: traversal(null_graph(3), 0) # null graph with several nodes
[3]: [0]
```

```
[4]: traversal(path_graph(4), 1) # start from node 1
[4]: [1, 0, 2, 3]
```

```
[5]: traversal(complete_graph(4), 0)
[5]: [0, 1, 2, 3]
```

As you can check, only the reachable nodes are visited and they're visited once, even if there are multiple paths from the start node to it.

Exercise 17.7.1

Outline an algorithm that does a traversal to decide if a given non-empty undirected graph is connected.

Hint Answer

17.7.4 Second algorithm

In a rooted tree, there's a single path from the root to each node. A traversal of a rooted tree can simply return a sequence of visited nodes, like the algorithm above, because we know exactly how each node was reached: via the edge from its parent.

But for graphs in general, a node may be reached in several ways, via any of its neighbours. It may be more useful to return the actually traversed **subgraph**, i.e. the subset of nodes visited and edges followed.

The next algorithm is a modification of the first so that we can see how a graph is traversed. Instead of marking out-neighbours to be processed, I mark the outgoing edges. When picking the next edge to process, I add it to the subgraph, which initially has just the start node.

1. let *visited* be a digraph with node *start*
2. let *unprocessed* be the set of the outgoing edges from *start*
3. while *unprocessed* isn't empty:
 1. remove one edge (*previous, current*) from *unprocessed*
 2. if *visited* doesn't have node *current*:
 1. visit *current*
 2. add *current* to *visited*
 3. add (*previous, current*) to *visited*
 4. for each *neighbour* in out-neighbours of *current* in *graph*:
 1. add (*current, neighbour*) to *unprocessed*

Why does the if-statement not add node *previous* to graph *visited*?

An edge (A, B) is added to *unprocessed* by step 2 or by step 3.2.4.1. Either step is preceded by adding A to the graph, in step 1 or step 3.2.2. Hence, when removing (A, B) from *unprocessed*, i.e. when following the edge from A to B, we only have to add B to the graph.

Changing the type of *visited* from a set to a digraph doesn't affect the algorithm's complexity. Every step on *visited* (adding a node, checking if it has a node) still takes constant time. Keeping pairs of nodes in *unprocessed* instead of single nodes doesn't change the complexity either.

The code is as follows.

```
[6]: def traversed(graph: DiGraph, start: Hashable) -> DiGraph:
    """Return the traversed subgraph when beginning at start.

    Preconditions: graph.has_node(start)
    """
    visited = DiGraph()
    visited.add_node(start)
    unprocessed = set()
    for neighbour in graph.out_neighbours(start):
        unprocessed.add((start, neighbour))
    while len(unprocessed) > 0:
        edge = unprocessed.pop()
        previous = edge[0]
        current = edge[1]
        if not visited.has_node(current):
            visited.add_node(current)
            visited.add_edge(previous, current)
```

(continues on next page)

(continued from previous page)

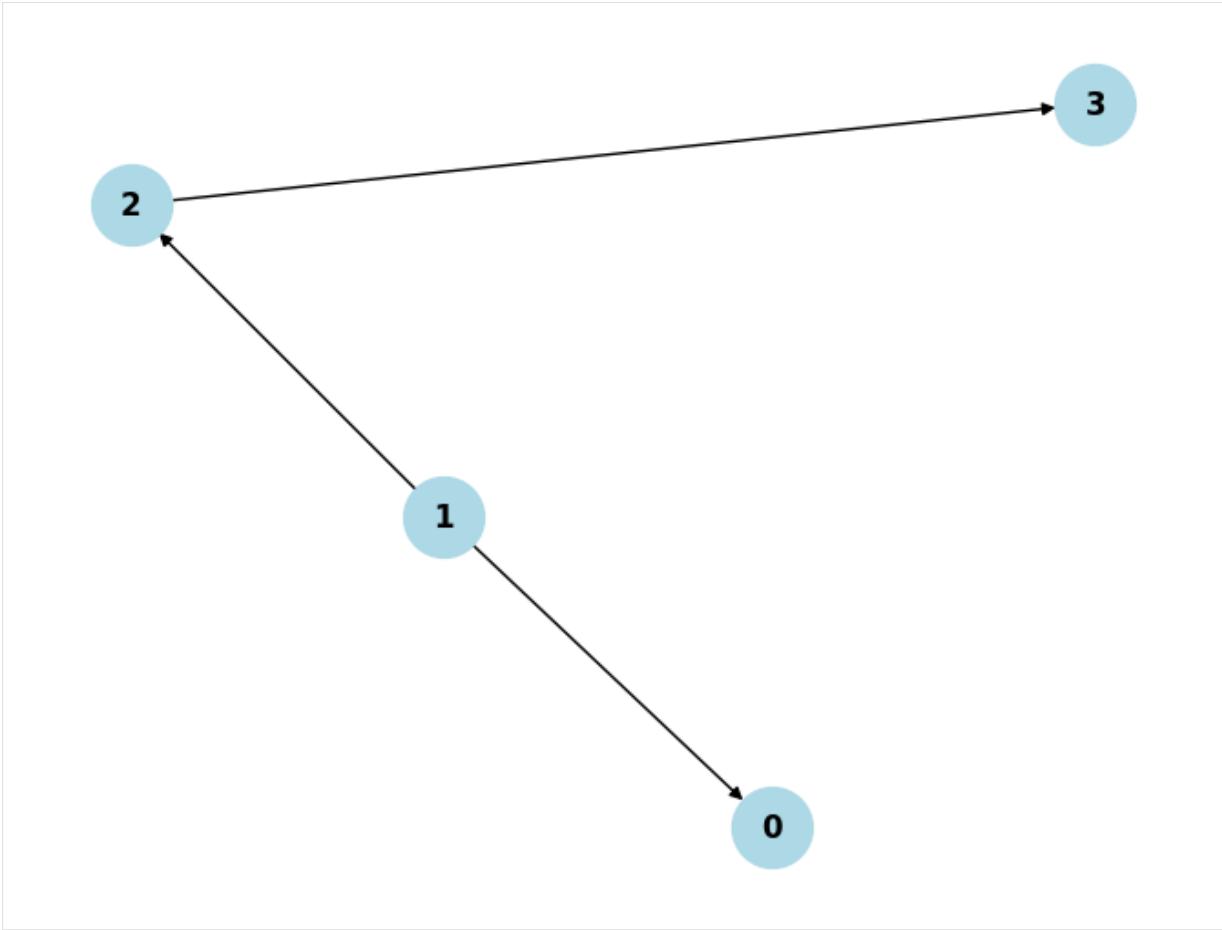
```
for neighbour in graph.out_neighbours(current):
    unprocessed.add((current, neighbour))
return visited
```

Let's see the traversal in action, including with a random graph.

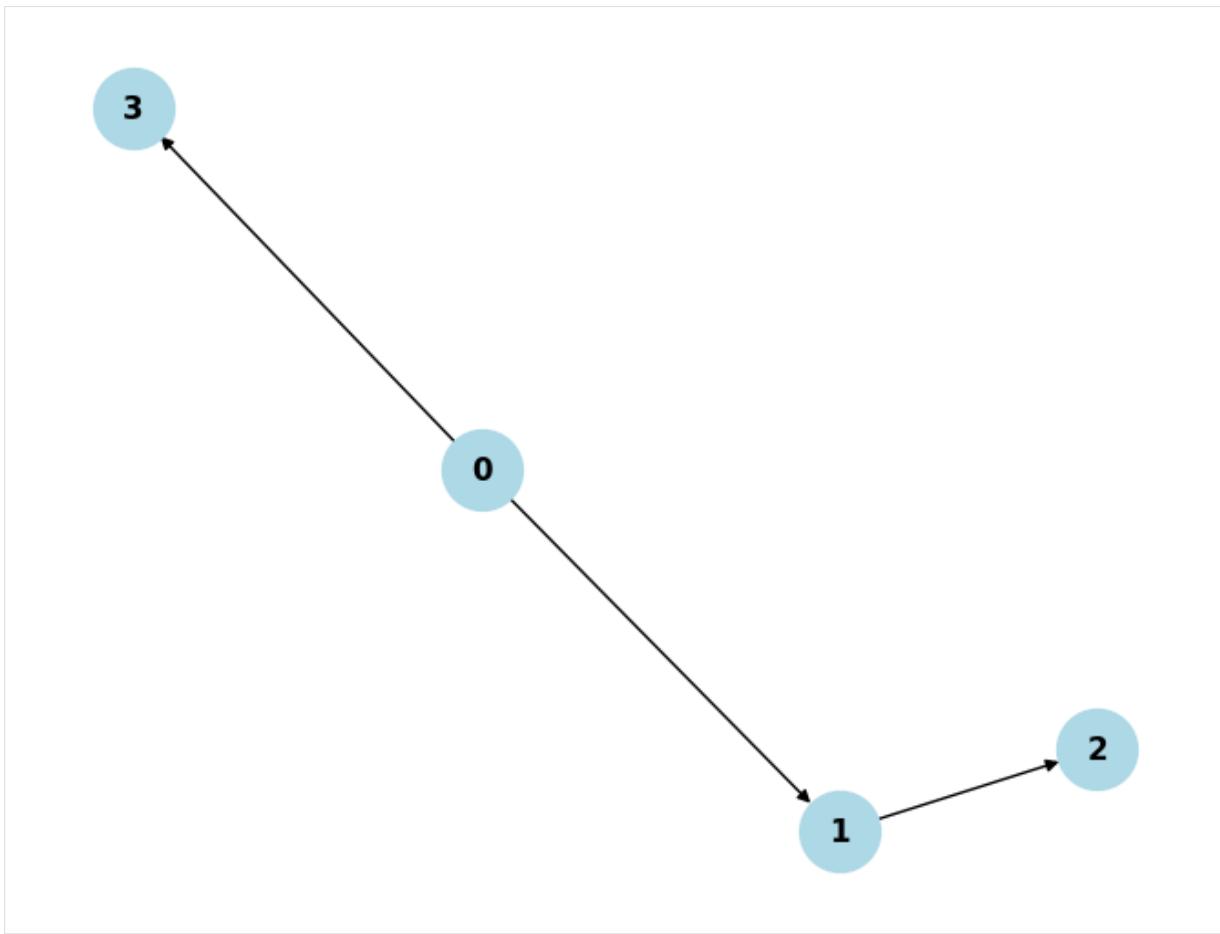
```
[7]: traversed(null_graph(3), 0).draw() # null graph with several nodes
```



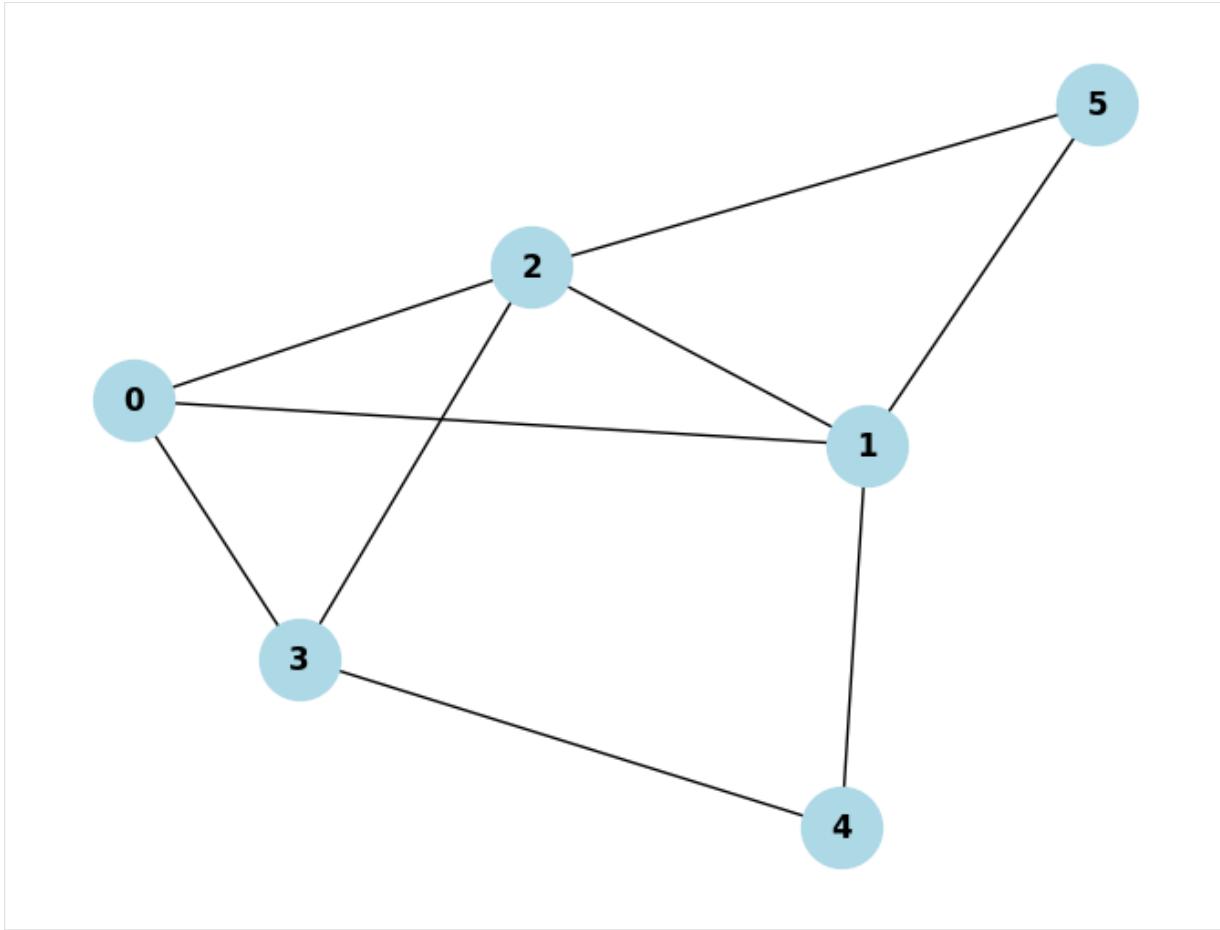
```
[8]: traversed(path_graph(4), 1).draw() # start from node 1
```



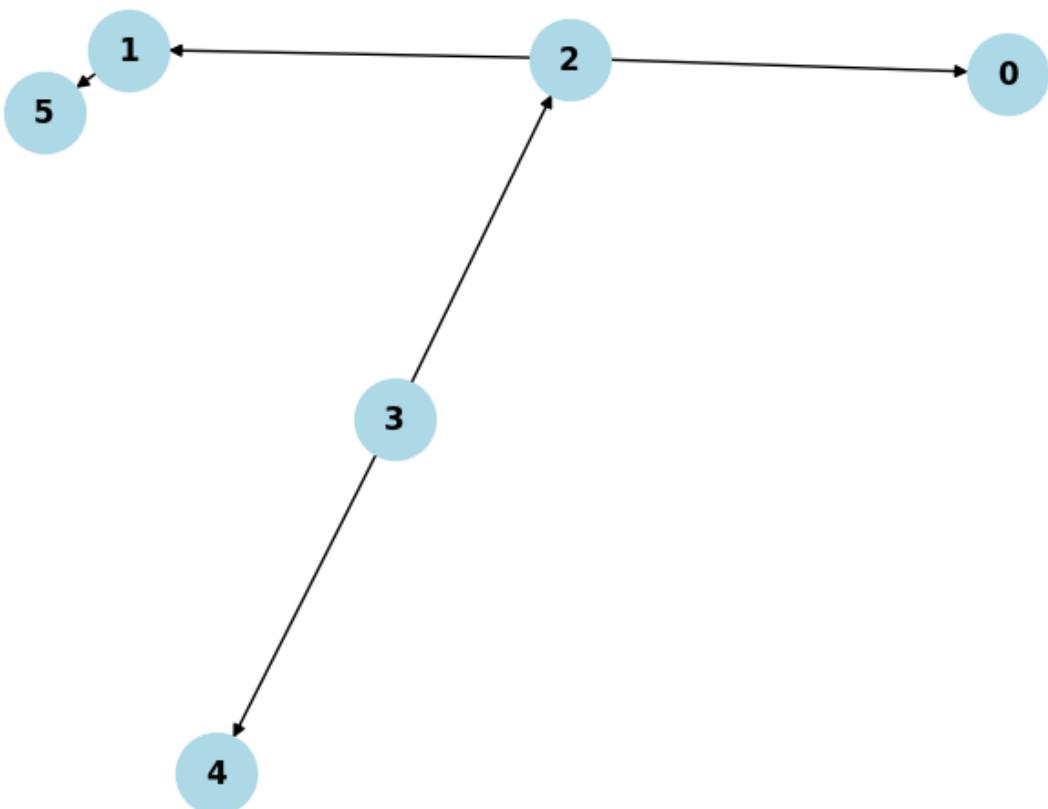
```
[9]: traversed(complete_graph(4), 0).draw()
```



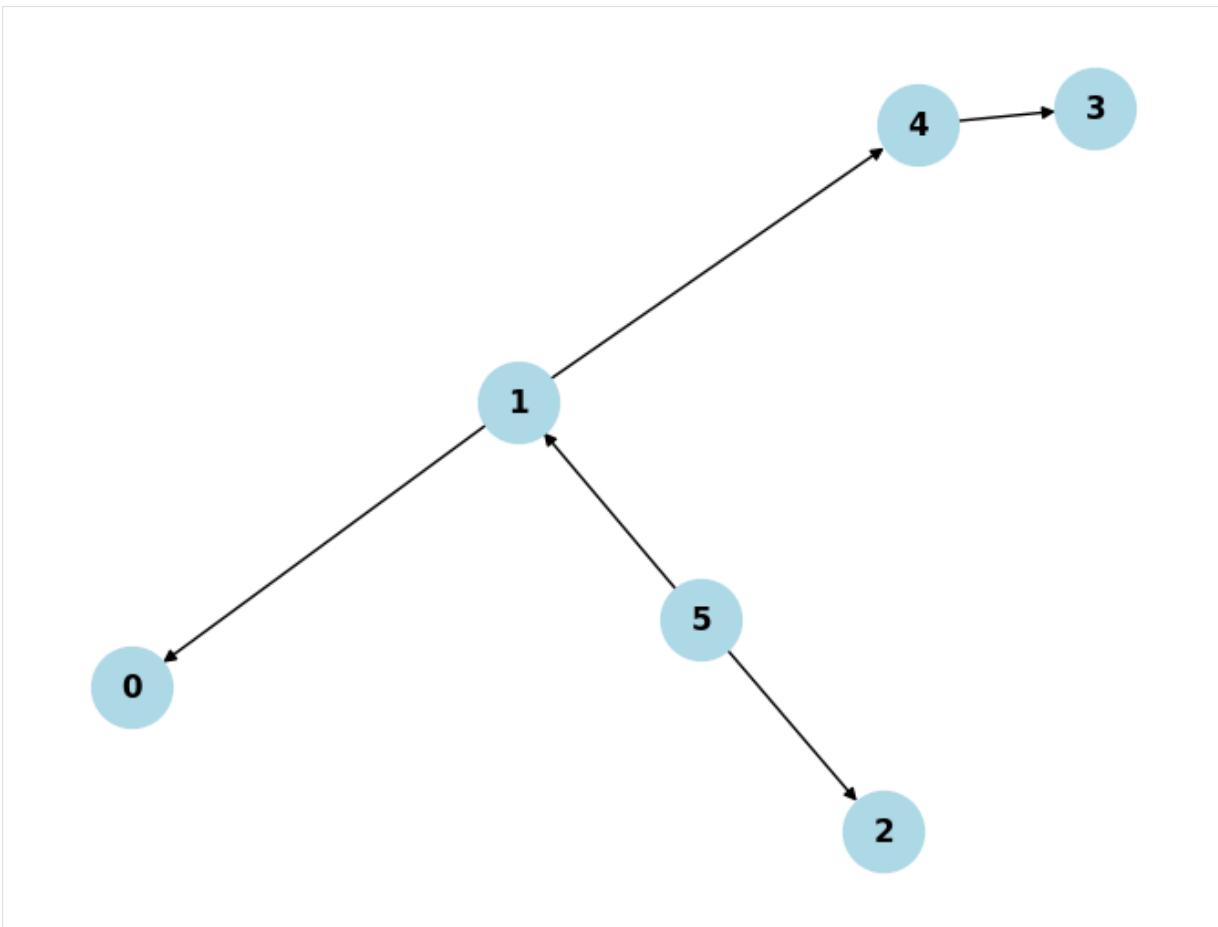
```
[10]: random = random_graph(6, 0.4)
random.draw()
```



```
[11]: traversed(random, 3).draw() # start from node 3
```



```
[12]: traversed(random, 5).draw() # start from node 5
```



The generated digraph has one edge fewer than the number of nodes because for every node, except the start, exactly one of its incoming edges is followed to visit it. By construction, every node is reachable from the start node, so the generated digraph is connected. These two facts mean that there's a single path from the start node to each other node. Hence there are no cycles: the generated graph is a DAG. If we ignore edge directions, then we obtain a tree, rooted at the start node.

Exercise 17.7.2

Outline an algorithm that does a traversal to decide if a given non-empty undirected graph is a tree.

Hint Answer

17.8 Breadth- and depth-first search

We saw two kinds of *traversals for rooted trees*. A breadth-first search (BFS) proceeds from the root downwards, level by level: it first visits the root, then its children, then their children, and so on. A depth-first search (DFS) goes down each subtree as far as it can before exploring the next one.

We can traverse graphs in similar ways. Since a traversal finds all nodes reachable from a given node, traversals can also be seen as searches.

Let's first get the usual preamble out of the way.

```
[1]: %run -i ../m269_digraph  
%run -i ../m269_ungraph  
%run -i ../m269_graphs
```

17.8.1 Breadth-first search

Our traversal algorithm puts the unprocessed edges in a set and picks one of them to visit the next node. The traversal is a rambling walk that visits nodes in no particular order.

However, a BFS traverses a graph in a very specific order: it visits first the start node, then its out-neighbours, then their out-neighbours, and so on. To ensure this, we simply process the edges in the order they're found. The collection of unprocessed edges must be a first-in, first-out sequence. The BFS algorithm is our traversal algorithm with a small change: we use a queue of unprocessed edges instead of a set.

Like sets, queues can support the addition and removal of items in constant time, so the complexity remains unchanged.

To write the code, I copy the `traversed` function, import our implementation of queues, and make small changes to the five lines involving the variable `unprocessed`.

```
[2]: %run -i ../m269_queue.py
```

```
[3]: # this code is also in m269_digraph.py
```

```
def bfs(graph: DiGraph, start: Hashable) -> DiGraph:  
    """Return the subgraph traversed by a breadth-first search.  
  
    Preconditions: graph.has_node(start)  
    """  
    # changes from traversed function noted in comments  
    visited = DiGraph()  
    visited.add_node(start)  
    unprocessed = Queue()  # was set  
    for neighbour in graph.out_neighbours(start):  
        unprocessed.enqueue((start, neighbour))  # was add()  
    while unprocessed.size() > 0:  # was len()  
        edge = unprocessed.dequeue()  # was pop()  
        previous = edge[0]  
        current = edge[1]  
        if not visited.has_node(current):  
            visited.add_node(current)  
            visited.add_edge(previous, current)  
            for neighbour in graph.out_neighbours(current):  
                unprocessed.enqueue((current, neighbour))  # was
```

(continues on next page)

(continued from previous page)

```
→add()
return visited
```

17.8.2 Depth-first search

A depth-first search first visits a node A, then one of A's out-neighbours, let's call it B, then one of B's out-neighbours, and so on. To obtain this behaviour, after adding the outgoing edges of a visited node, we must follow one of them. It is simplest to follow the last added edge. The collection of unprocessed edges must be a last-in, first-out sequence. So, by changing the traversal algorithm to use a stack of unprocessed edges instead of a set, we obtain the DFS algorithm.

[4]: %run -i ./m269_stack

Like for sets, adding and removing items from a stack takes constant time, so again the complexity isn't affected.



Note: A breadth- or depth-first search of a graph has worst-case complexity $\Theta(n + e)$.

The code is the same as for `bfs`, but with `unprocessed` being a stack rather than a queue.

[5]: # this code is also in m269_digraph.py

```
def dfs(graph: DiGraph, start: Hashable) -> DiGraph:
    """Return the subgraph traversed by a depth-first search.

    Preconditions: graph.has_node(start)
    """
    visited = DiGraph()
    visited.add_node(start)
    unprocessed = Stack()  # was Queue()
    for neighbour in graph.out_neighbours(start):
        unprocessed.push((start, neighbour))  # was enqueue()
    while unprocessed.size() > 0:
        edge = unprocessed.pop()  # was dequeue()
        previous = edge[0]
        current = edge[1]
        if not visited.has_node(current):
            visited.add_node(current)
            visited.add_edge(previous, current)
            for neighbour in graph.out_neighbours(current):
                unprocessed.push((current, neighbour))  # was enqueue()
```

(continues on next page)

(continued from previous page)

```
→enqueue()  
return visited
```

17.8.3 Tests

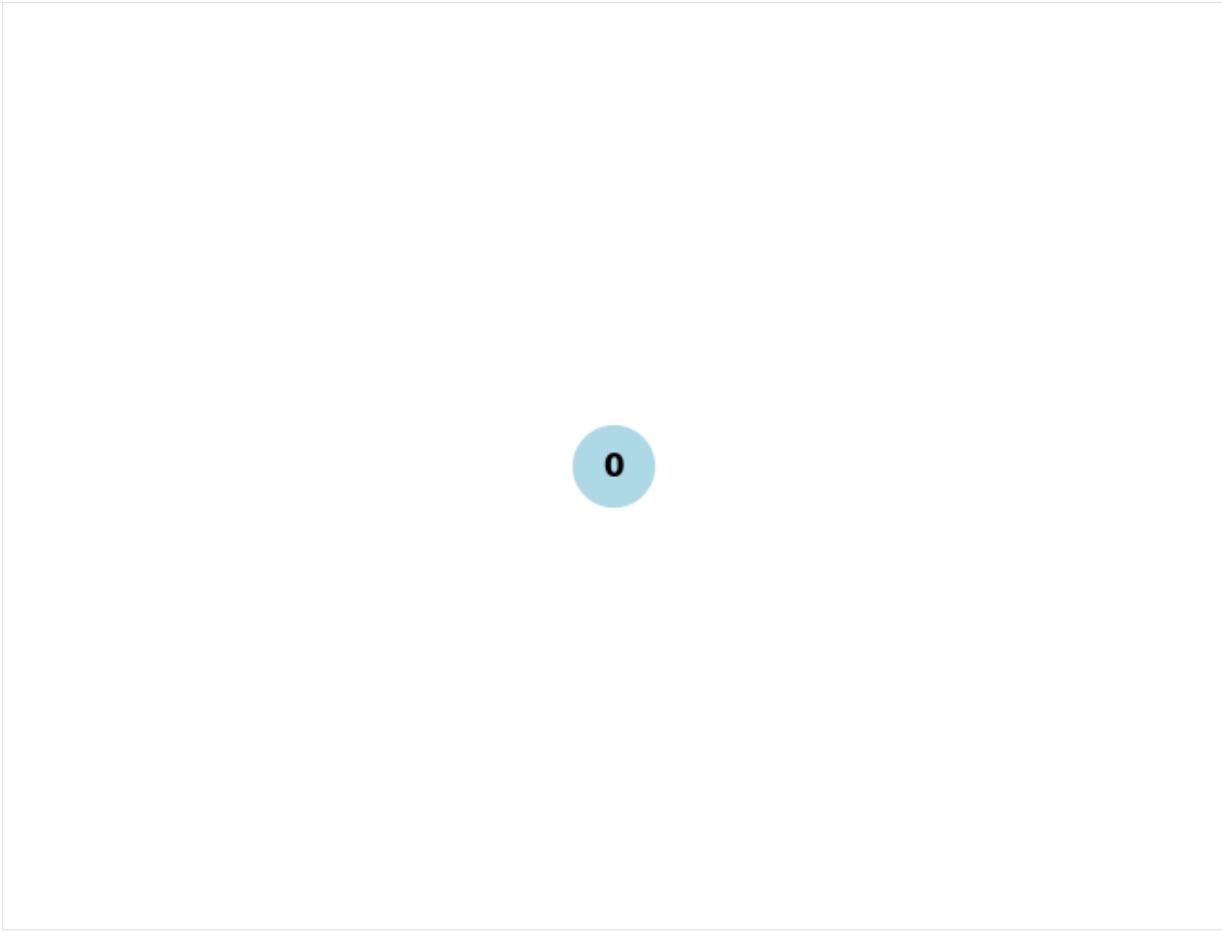
Let's compare the subgraphs generated by BFS and DFS when traversing the same input graph from the same start node.

First, a null graph with multiple nodes. BFS and DFS generate the same graph, just with the start node, because there are no edges to follow.

```
[6]: bfs(null_graph(3), 0).draw() # nodes: 0, 1, 2; start node: 0
```

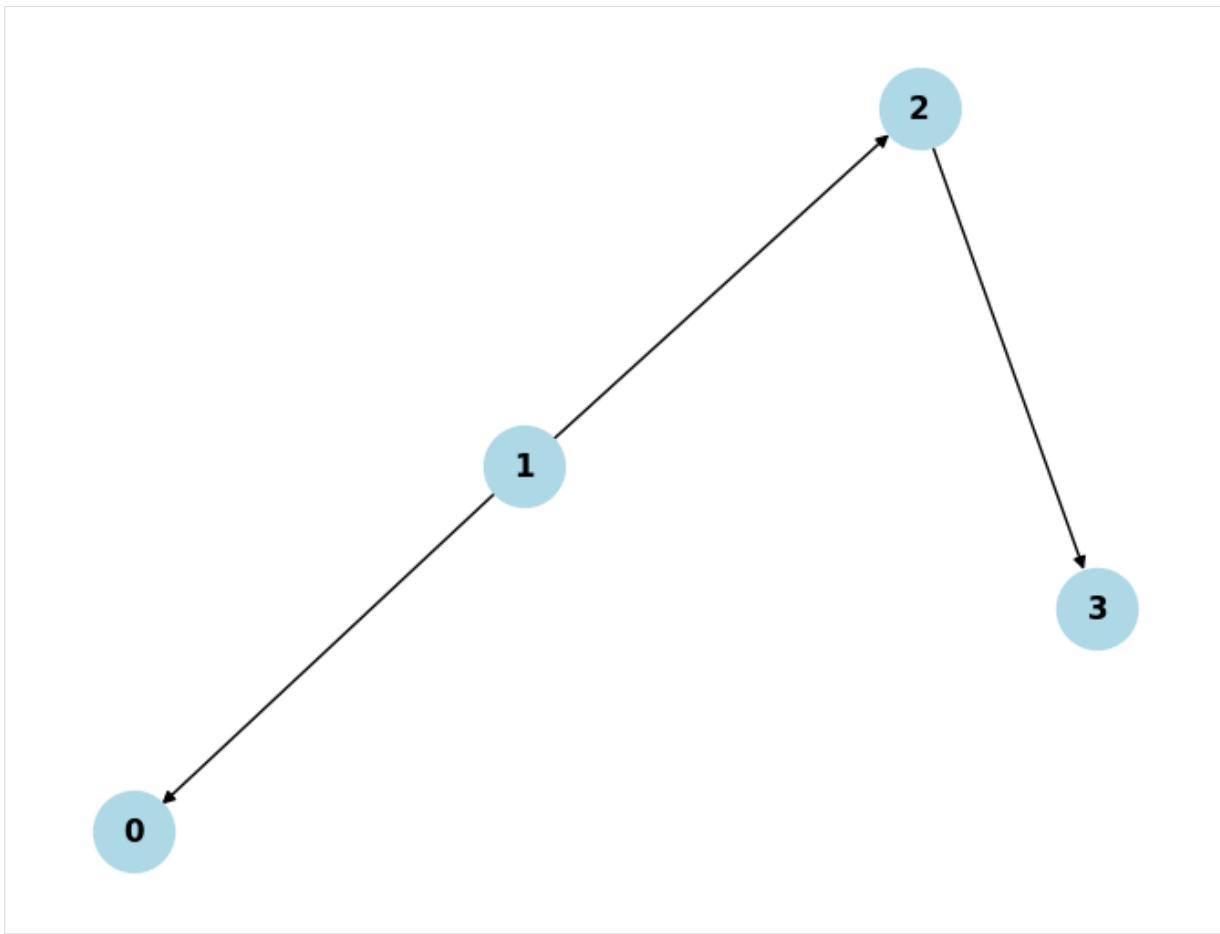


```
[7]: dfs(null_graph(3), 0).draw()
```

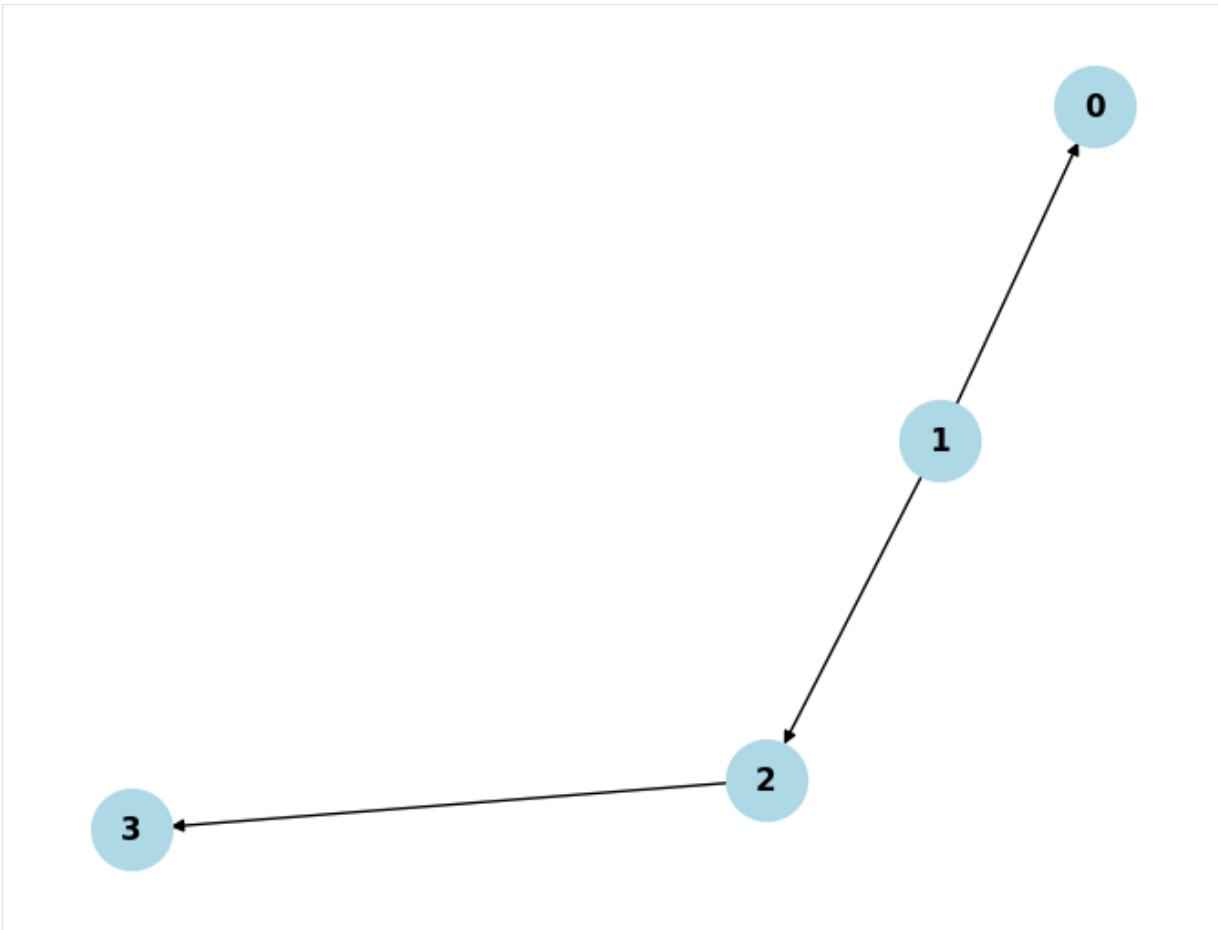


BFS and DFS also traverse path graphs in the same way, as there's no choice of which edges to follow.

```
[8]: bfs(path_graph(4), 1).draw() # start node: 1
```

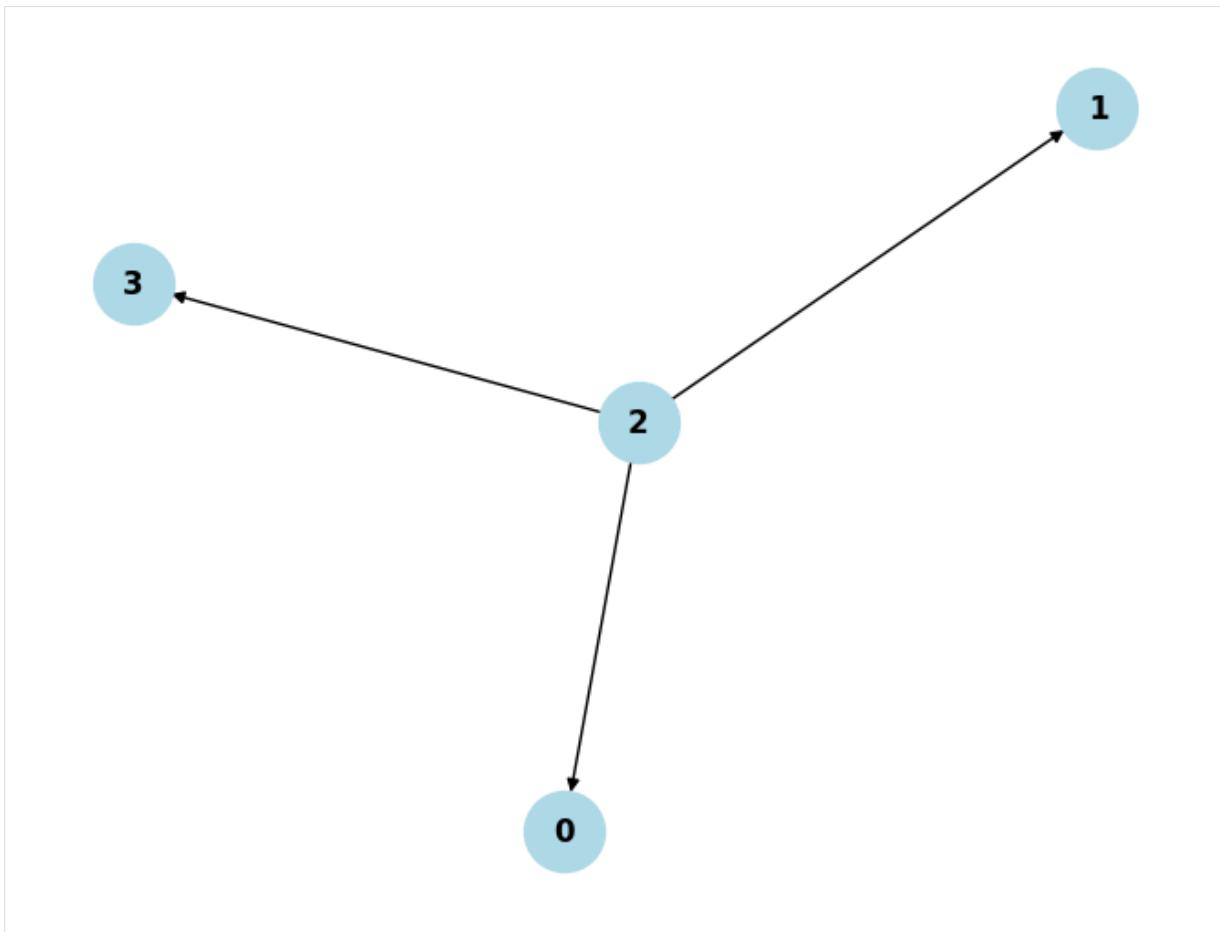


```
[9]: dfs(path_graph(4), 1).draw()
```

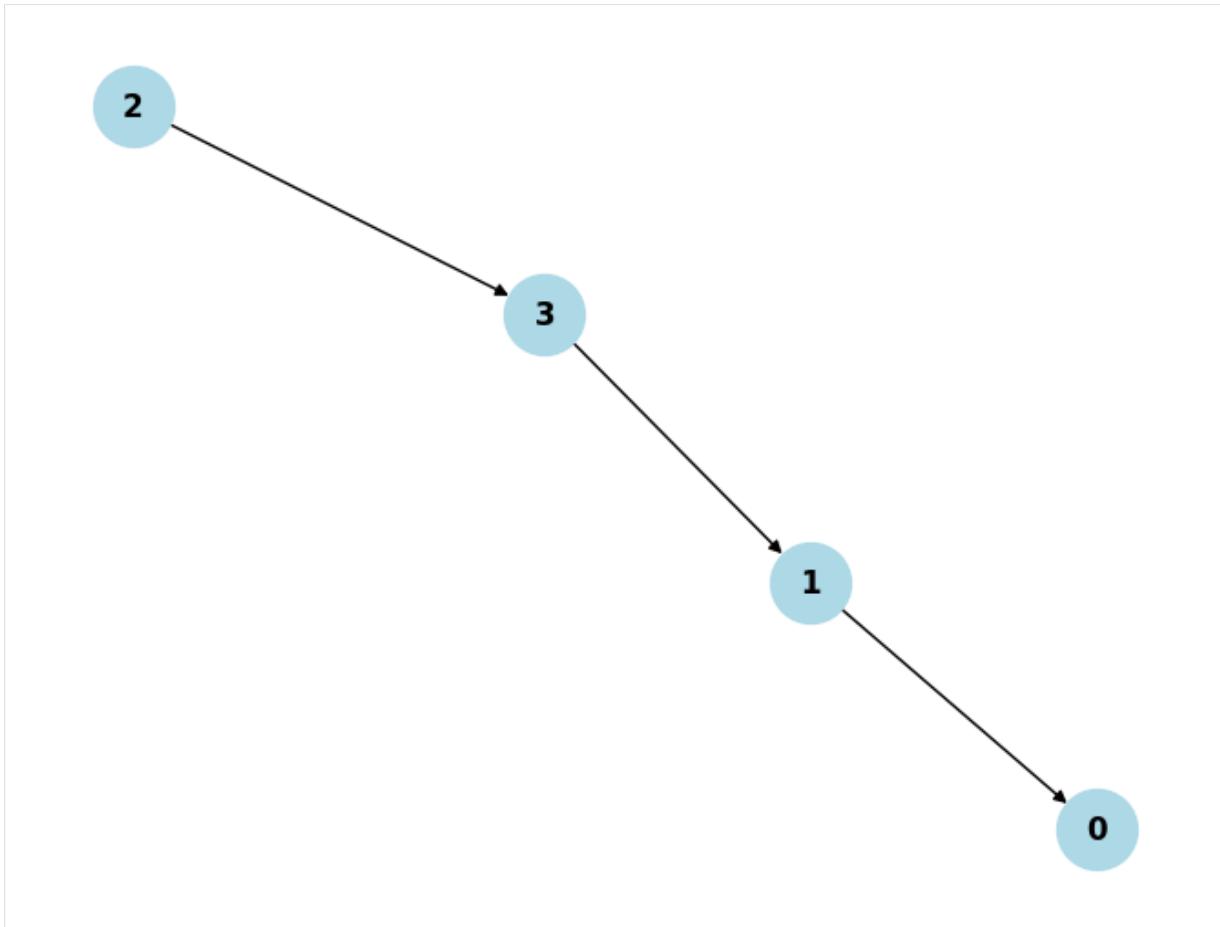


However, for complete graphs, the outputs differ. BFS generates a star-shaped graph because all other nodes are out-neighbours of the start node. DFS generates a directed path graph, because there's always an edge that can be followed from any current node to any yet-unvisited node.

```
[10]: bfs(complete_graph(4), 2).draw() # start node: 2
```

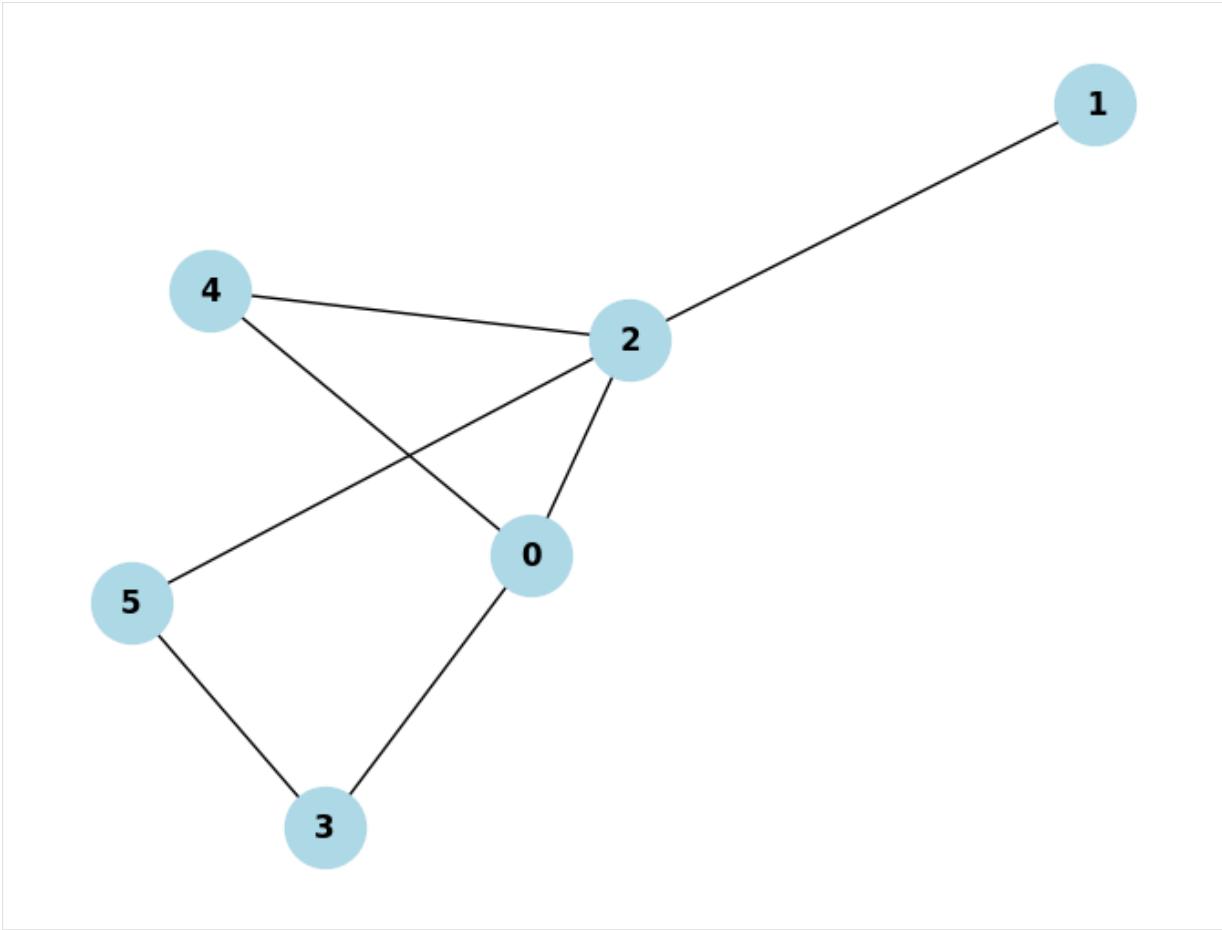


```
[11]: dfs(complete_graph(4), 2).draw()
```

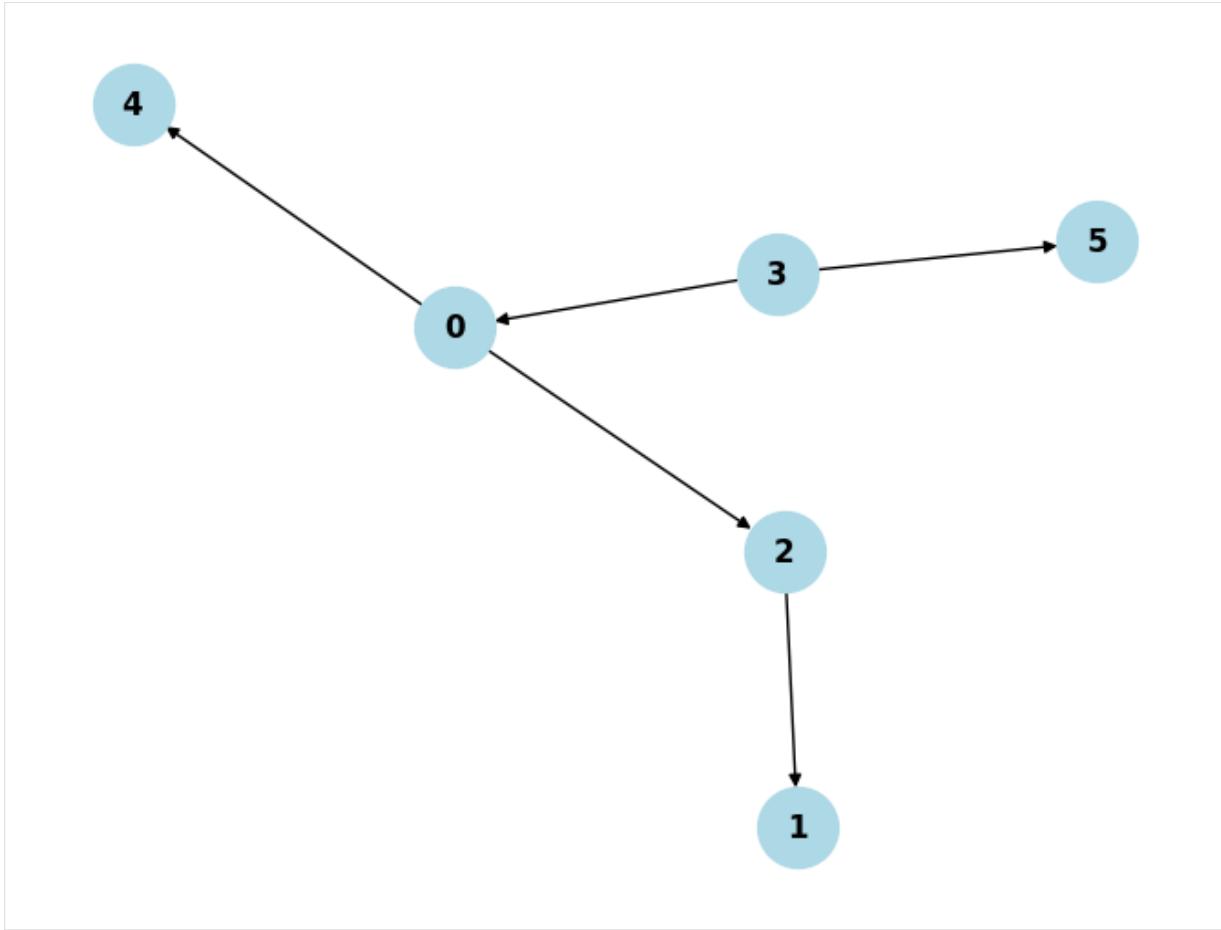


For random graphs, the BFS and DFS traversals may be different or the same, as you can observe by executing the following cells several times. I set the edge probability to 40% but you can change it. As the probability increases from 0% (null graph) to 100% (complete graph), the BFS and DFS traversals are less likely to be the same: the traversals are always equal for null graphs and always different for complete graphs, no matter the start node.

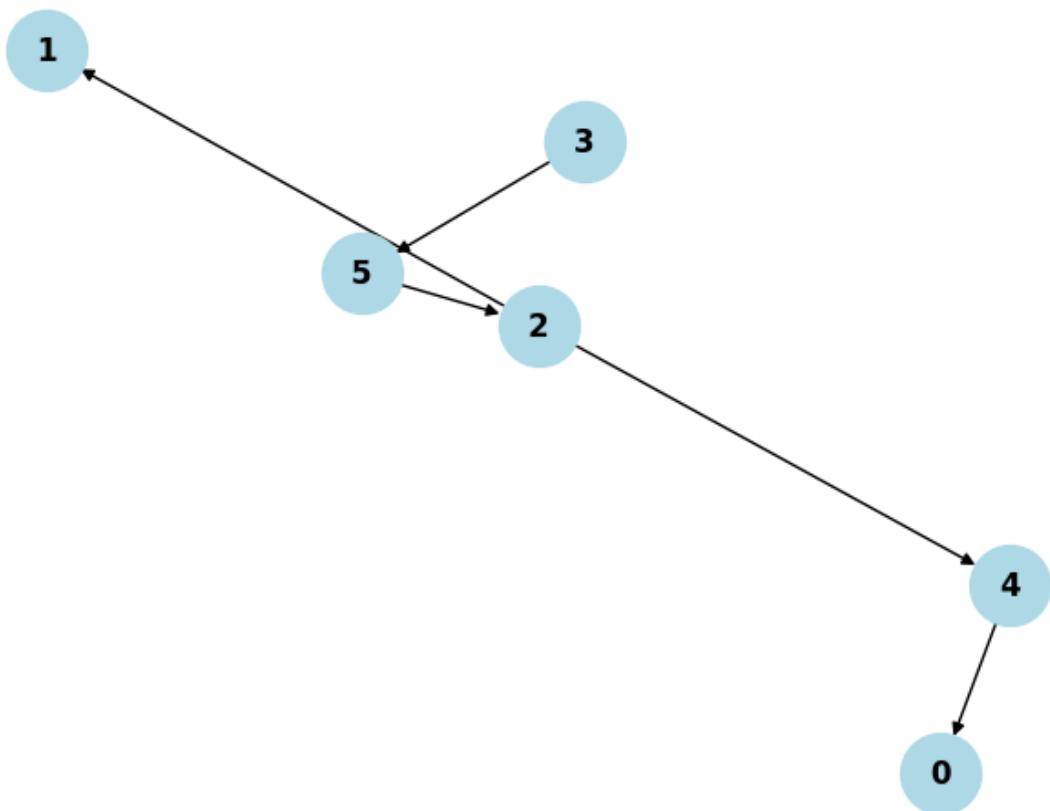
```
[12]: random = random_graph(6, 0.4) # set a lower or higher probability  
random.draw()
```



```
[13]: bfs(random, 3).draw() # start node: 3
```



```
[14]: dfs(random, 3).draw()
```



17.8.4 Comparison

Like the basic ‘walk’ traversal, BFS and DFS produce a directed tree of the nodes reachable from the start. The three algorithms traverse an acyclic subgraph of the input graph and thus any of them can be used to decide if the input graph is connected or is a tree.



Note: If a graph problem is about connectivity, you can probably solve it with any traversal algorithm.

However, the difference between DFS and BFS leads to one advantage of BFS. DFS extends one path as far as it can. When that path leads to no unvisited nodes, it tries a different path. BFS instead extends different paths from the start node one edge at a time, because it first visits the start node’s out-neighbours, then their out-neighbours, and so on. In other words, BFS first finds the nodes one edge away from the start, then those two edges away, etc.



Note: BFS finds the shortest paths from the start node to every other reachable node.

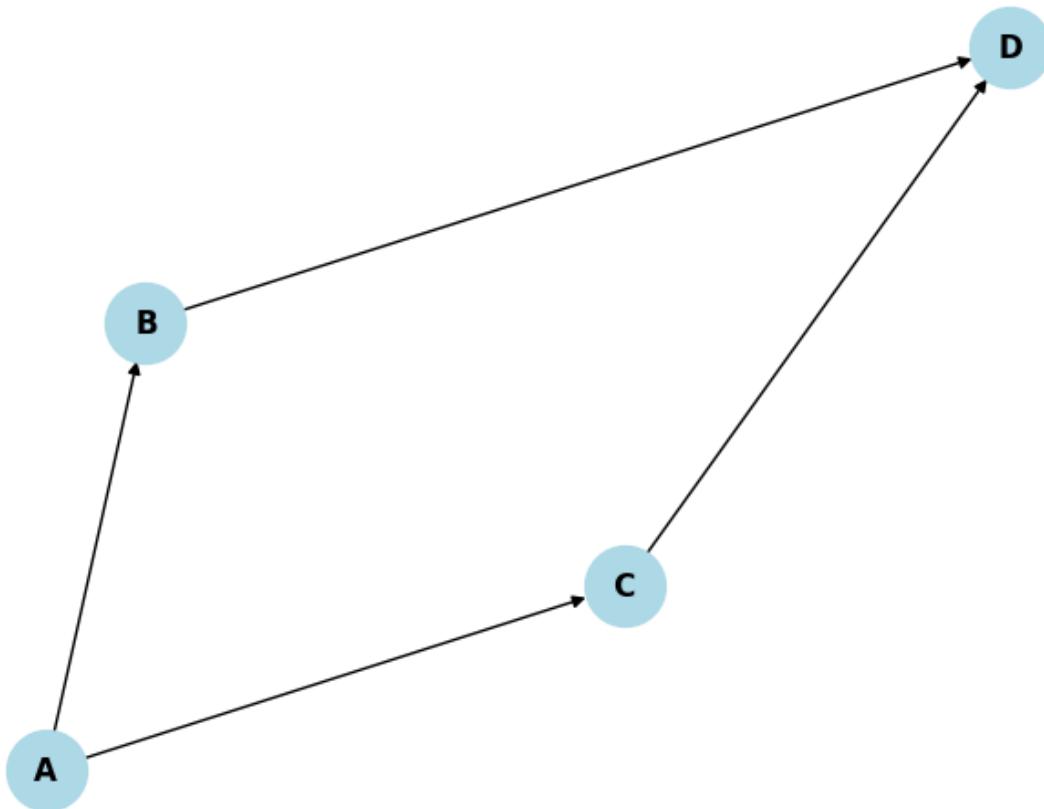
There may be several BFS and DFS subgraphs for the same graph and start node, but the `bfs` and `dfs` functions only produce one of them. Which one is produced depends on the order in which nodes are stored internally in the sets of neighbours.

For example, a complete graph with n nodes has $(n - 1)!$ DFS subgraphs from each start node. After the start node, DFS can visit any of the $n - 1$ other nodes, then any of the $n - 2$ remaining nodes, and so on. The above 4-node complete graph has $(4 - 1)! = 6$ DFS subgraphs from start node 0: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$, $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ and every other directed path graph obtained by permutation of nodes 1, 2 and 3.

However, there's a single BFS subgraph for a complete graph and start node: as seen earlier, it's in the form of a star, with the start node in the centre.

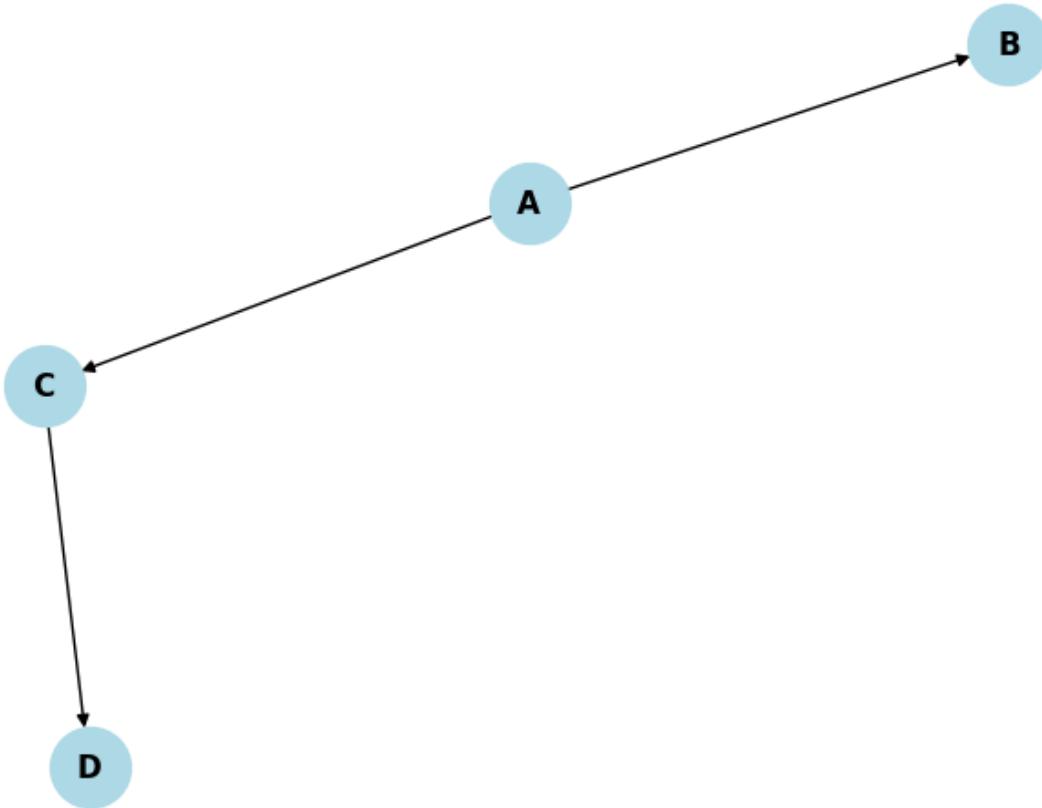
The simplest example of a graph with two BFS subgraphs is the following:

```
[15]: kite = DiGraph()
for node in "ABCD":
    kite.add_node(node)
for edge in ("AB", "AC", "BD", "CD"):
    kite.add_edge(edge[0], edge[1])
kite.draw()
```



There are two subgraphs starting from A, depending on whether D is visited from B or from C, but the function produces only one of them.

```
[16]: bfs(kite, "A").draw()
```



If you run the last cell, you may get the other subgraph. To understand why the same code can produce different outputs at different times for the same input, remember that in Python the hash value of a string changes in every interpreter session (see the paragraph before [Section 8.3.3](#)) and sets are implemented with hash tables. So, the internal order of strings ‘B’ and ‘C’ in the set of A’s neighbours may change between sessions, leading to a different BFS subgraph.

17.9 Summary

A **graph** consists of a set of items called **nodes** and a set of pairs of nodes, called **edges**. Each edge connects a node with a different node and can be directed or undirected. A **directed graph** (or **digraph**) only has directed edges; an **undirected graph** only has undirected edges. Between each pair of nodes, there’s at most one undirected edge or two directed edges in opposing directions.

A relation between pairs of items is a **binary relation**. A binary relation is **symmetric** if it’s reciprocal: if A is related to B, then B is related to A.

Since edges connect pairs of nodes, graphs can model any set of entities and a binary relation over them. This includes many kinds of networks. We can use an undirected graph only if the relation is symmetric.

17.9.1 Terminology

Two nodes are **adjacent** if connected by an edge. If there's an edge from A to B, then A is an **in-neighbour** of B and B is an **out-neighbour** of A. An undirected edge is both from A to B and from B to A.

A node's **neighbours** are its adjacent nodes. The **degree** of a node is the number of edges attached to it.

The **in- and out-degree** of a node are the number of its in- and out-neighbours, respectively. In an undirected graph, the in-neighbours and the out-neighbours of a node are the same, so its in- and out-degrees are the same as the degree.

Every undirected graph can be represented as a digraph with twice the edges, by replacing each undirected edge with two opposing directed edges. Algorithms based on in- and out-neighbours and in- and out-degrees can thus be applied to directed *and* undirected graphs.

A **path** of length k from node A to node B is a sequence of $k + 1$ nodes, starting with A and ending with B. There must be an edge from each node in the sequence to the next one. In addition, all nodes and edges in a path must be different. Node B is **reachable** from node A if there's a path from A to B. A **shortest path** has the fewest possible edges among all paths from A to B.

An undirected graph is **connected** if all pairs of nodes are mutually reachable; otherwise it's **disconnected**. If there's a node S from which all other nodes are reachable, then all nodes are mutually reachable via S: the graph is connected. A digraph is connected or disconnected if we obtain a connected or disconnected undirected graph when ignoring all edge directions.

A **cycle** is a path with the exception that the first and last node are the same. A graph with at least one cycle is **cyclic**; otherwise it's **acyclic**. The acronym **DAG** stands for directed acyclic graph. A **tree** is a connected acyclic undirected graph. In a tree, there's a single path from one node to any other node. A tree with n nodes has $n - 1$ edges.

The **empty graph** has neither nodes nor edges. A **null graph** has nodes but no edges. A **path graph** is a non-empty undirected graph that can be laid out in a line, with each node except the last connected to the next one. If we take a path graph with at least three nodes and connect the last node to the first, we obtain a cycle graph. A **complete graph** has all possible edges. A **random graph** has each possible edge with the same probability.

A graph is **dense** if it has many of the possible edges; it's **sparse** if it has few of the possible edges. There's no precise definition to classify each graph as sparse, dense or neither. Complete graphs are the densest graphs; null graphs are the sparsest graphs.

A **subgraph** of a graph G consists of only a subset of the nodes and edges of G.

17.9.2 Data structures

An **edge list** representation consists of one collection containing the nodes and one collection containing the edges. Several data structures are possible, depending on the type of the collections, e.g. sequences or sets, and how they are stored, e.g. as arrays, linked lists, lookup tables or hash tables.

An **adjacency matrix** representation consists of an n by n matrix that represents all possible

edges and a lookup table that maps the indices to nodes. The cell in the row for node A and in the column for node B is a Boolean indicating if there's an edge from A to B. Adjacency matrices waste memory for sparse graphs and for undirected graphs.

An **adjacency list** representation consists of a map where the keys are nodes and the values are their out-neighbours (and possibly their in-neighbours too). Each value is the adjacency list for the corresponding node. Several data structures are possible to store the map and each adjacency list. For example, the map can be implemented with a lookup table, hash table or BST, and an adjacency list can be represented as a sequence or a set.

Of all these possibilities, the most efficient for most graphs is an adjacency list representation in which the map and the adjacency lists are stored in hash tables. This requires the representation of nodes to be hashable.

In M269, the undirected and directed graph ADTs have the following operations with the listed worst-case complexities for undirected and directed graphs. When referring to graphs, we use n for the number of nodes and e for the number of edges.

Operation	Written as	Undirected	Directed
new	let G be an empty graph	$\Theta(1)$	$\Theta(1)$
has node	$A \in G$	$\Theta(1)$	$\Theta(1)$
add node	add A to G	$\Theta(1)$	$\Theta(1)$
remove node	remove A from G	$\Theta(n)$	$\Theta(n)$
has edge	$(A, B) \in G$	$\Theta(1)$	$\Theta(1)$
add edge	add (A, B) to G	$\Theta(1)$	$\Theta(1)$
remove edge	remove (A, B) from G	$\Theta(1)$	$\Theta(1)$
nodes	nodes of G	$\Theta(n)$	$\Theta(n)$
edges	edges of G	$\Theta(e)$	$\Theta(e)$
in-neighbours	in-neighbours of A in G	$\Theta(\text{degree}(A))$	$\Theta(n)$
out-neighbours	out-neighbours of A in G	$\Theta(\text{degree}(A))$	$\Theta(\text{out-degree}(A))$
neighbours	neighbours of A in G	$\Theta(\text{degree}(A))$	$\Theta(n)$
in-degree	in-degree of A in G	$\Theta(1)$	$\Theta(n)$
out-degree	out-degree of A in G	$\Theta(1)$	$\Theta(1)$
degree	degree of A in G	$\Theta(1)$	$\Theta(n)$

17.9.3 Traversals

A **traversal** of a graph starts from a given node and follows the outgoing edges to visit every node reachable from the start. The algorithm keeps a collection of yet-unprocessed edges.

While the collection isn't empty, it removes one edge from the collection. If it leads to a yet-unvisited node, the edge is followed, the node is visited, and its outgoing edges are added to the unprocessed collection.

If the collection is a set, the traversal visits nodes in no particular order.

If the collection is a FIFO sequence (queue), the nodes are visited in breadth-first order: first the out-neighbours of the start node, then their out-neighbours, and so on. This finds the shortest paths from the start node to each other reachable node.

If the collection is a LIFO sequence (stack), the nodes are visited in depth-first order: first one out-neighbour of the start node, then one of its out-neighbours, etc.

All three variants traverse an acyclic subgraph of the input graph. They all take constant time in the best case and $\Theta(n + e)$ in the worst case. They can all be used to decide if a non-empty undirected graph is connected and, in particular, a tree.

17.9.4 Python

Class `Hashable` in module `typing` can be used to annotate function parameters as being hashable objects.

Function `random` in module `random` returns each time it's called a random floating-point number from 0 (inclusive) to 1 (exclusive).

Replacing `self.` with `super()`. allows a subclass to call methods of its superclass.

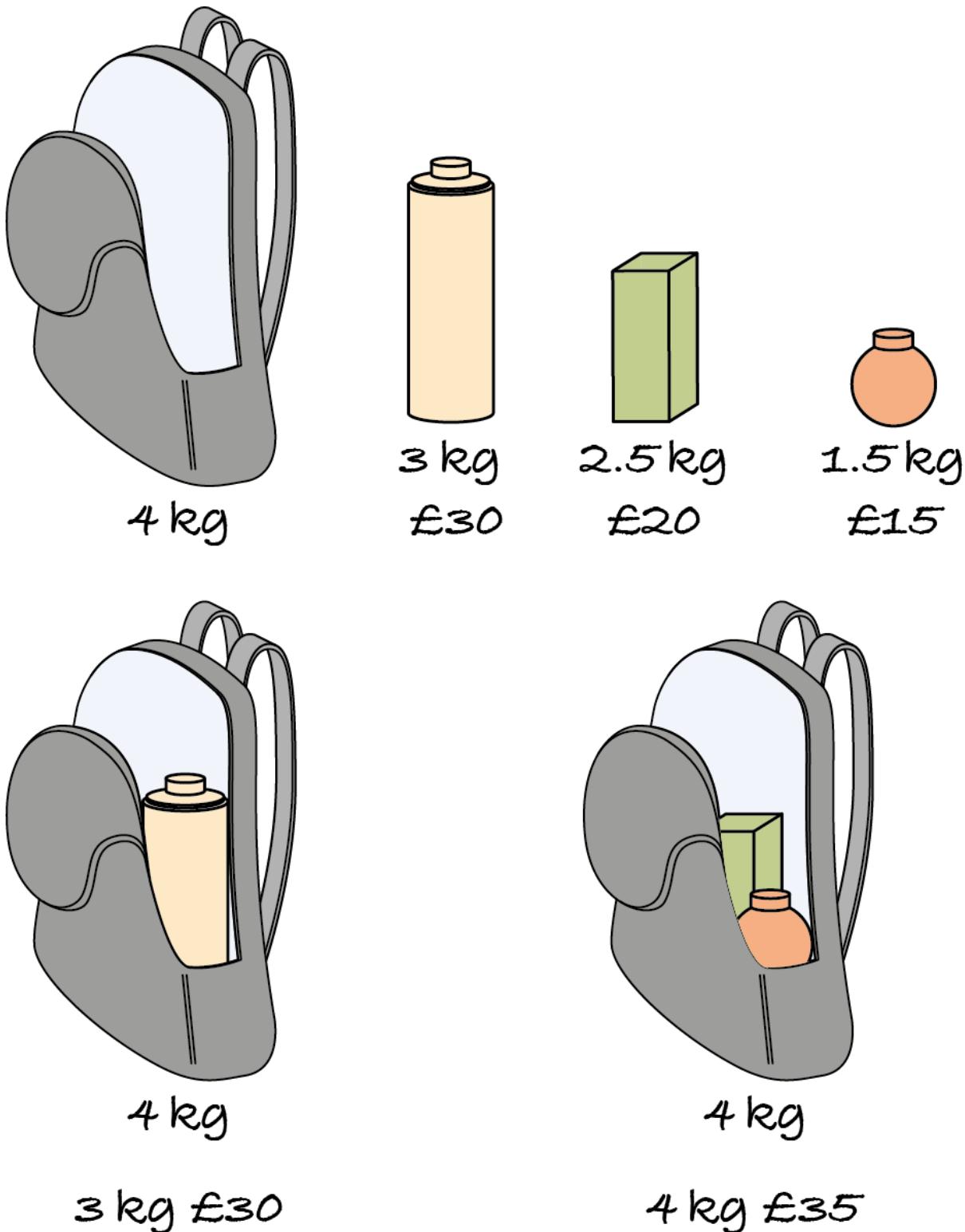
CHAPTER 18

GREED

Consider again the *0/1 knapsack problem*: given the weights and values of a set of items, we want a most valuable subset of items we can pack in the knapsack, without exceeding its weight capacity. We solved this optimisation problem with an exhaustive search that generates all possible subsets and, for each one, checks if it fits the knapsack and has the largest value found so far.

An alternative approach is to start with an empty knapsack and keep adding the most valuable remaining item that still fits in the knapsack. This algorithm is highly efficient: it sorts the items by descending value in log-linear time and then goes through them in linear time. Each item is either put in the knapsack or skipped, because its weight exceeds the remaining capacity of the knapsack. Contrast this with the exhaustive search, which generates and tests an exponential number of subsets.

There's a snag, however. The second algorithm isn't correct: it doesn't generate a solution for some inputs as the following *counter-example* shows. The most valuable item is the heaviest. Choosing it won't leave space for anything else. Instead, the two cheaper items form the most valuable subset.



An algorithm that incrementally constructs the solution by always choosing the best available option, for some notion of ‘best’, is a **greedy algorithm**. As in real life, greed is often not a good approach. Greedy algorithms solve optimisation problems by choosing a local optimum at each step, in the hope of reaching a global optimum. For many problems, this doesn’t happen.

However, when it does work, greed is an excellent approach. Greedy algorithms don’t revisit the choices made, they simply ‘press ahead’, so they’re iterative, simple and efficient. Even when they don’t provide a correct solution, they can be used as *heuristic algorithms* that quickly

compute an approximation of the solution.

This chapter explains how to design greedy algorithms, shows problems for which they do compute the correct solution and problems for which they don't, with tips for finding counter-examples that show that a greedy algorithm is incorrect.

This chapter also introduces weighted graphs, in which edges have numbers to indicate the distance, cost or time needed to traverse them. Such graphs will be used to illustrate two greedy graph algorithms.

The examples and exercises in this chapter support the usual learning outcomes.

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – you will learn about weighted graphs and greedy algorithms.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – you will learn how to find counter-examples to prove a greedy algorithm doesn't work.
- Write readable, tested, documented and efficient Python code – this chapter introduces classes for weighted graphs.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

18.1 Interval scheduling

After a long lockdown, Alice is keen to attend as many music festivals as possible. She has the start and end date of each festival she'd like to attend. What is a largest subset of festivals she can attend?

Consider groups of people trying to book a particular room for their meetings. Given the start and end time of each meeting, what's the largest number of meetings that can be scheduled for that room?

In general, the **interval scheduling** problem is: given a set of time intervals, find a largest subset without overlaps. Two time intervals overlap if they have at least one time point in common, e.g. if the end time of one interval is part of the other interval.

To simplify, I will represent time points as natural numbers and time intervals as pairs of start–end points, e.g. (3, 5) is the interval starting at time point 3 and ending at time point 5.

Operation: largest interval schedule

Inputs: *intervals*, a set of pairs of natural numbers

Preconditions: for every pair $(start, end)$, $start < end$

Output: *schedule*, a set of pairs of natural numbers

Postconditions:

- *schedule* is a subset of *intervals*
- for any $(s_1, e_1) \neq (s_2, e_2)$ in *schedule*, either $e_1 < s_2$ or $e_2 < s_1$
- no subset of *intervals* satisfying condition 2 is larger than *schedule*

The second postcondition states that for any two intervals in the solution, one must end before the other starts. The third postcondition states that the output has as many intervals as possible.

The problem definition uses the word ‘Operation’ instead of ‘Function’ because we’re not defining a mathematical function: a set of intervals may lead to several largest schedules as the next table shows. An algorithm can return any of them.

Case	<i>intervals</i>	<i>schedule</i>
no intervals	{ }	{ }
one interval	{(9, 17)}	{(9, 17)}
no overlap	{(4, 5), (0, 1), (2, 3)}	{(4, 5), (0, 1), (2, 3)}
some overlap	{(6, 10), (9, 17), (0, 5)}	{(6, 10), (0, 5)} or {(9, 17), (0, 5)}
all overlap	{(3, 5), (5, 6), (2, 7)}	{(3, 5)} or {(5, 6)} or {(2, 7)}

18.1.1 The greedy approach

A greedy algorithm constructs the output collection one item at a time. It first finds all items that are compatible with the choices made so far. For the knapsack problem, it’s those items that still fit in the knapsack. For interval scheduling, it’s those intervals that don’t overlap any already scheduled interval. I will call those items the *extensions* to the current partial solution.

Once the algorithm has computed the extensions, it makes a **greedy choice**: it chooses the best extension, or a single best one if there are several equally good options. The criterion used by the greedy choice is the crux of the algorithm; I’ll come to it later.

The algorithm continues choosing a best item among those compatible with previous choices, until the solution is complete. This is certainly the case when no more items are available, but depending on the problem the solution may be completed earlier.

A general algorithmic pattern for greedy algorithms is:

1. let *solution* be an empty collection
2. repeat:
 1. let *extensions* be the possible next items compatible with *solution*
 2. if *extensions* isn’t empty:
 1. add one of the best of *extensions* to *solution*
 3. until *extensions* is empty

This algorithm is computing the extensions from scratch in each iteration. Often, we know the possible extensions in advance (they’re the input items) and we just want to add the next best one to the solution. I gave an example in the introduction: we take the next most valuable item and put it in the knapsack if it fits.

For such problems, a more efficient greedy algorithm constructs the extensions once, by sorting the input items from best to worst. It then keeps picking the next best item, checks if it’s compatible with the partial solution, and if so extends the solution with that item.

1. let *solution* be an empty collection

2. let *extensions* be the input items sorted from best to worst
3. for each *item* in *extensions*:
 1. if *item* is compatible with *solution*:
 1. add *item* to *solution*

The greedy algorithm I outlined in the introduction for the knapsack problem is:

1. let *solution* be the empty set
2. let *extensions* be the input items sorted from most to least valuable
3. for each *item* in *extensions*:
 1. if weight of *item* + weight of *solution* \leq capacity of knapsack:
 1. add *item* to *solution*



Note: Greedy algorithms sometimes involve sorting the input.

Exhaustive search generates all possible candidates (unless it can stop early), to make sure it finds the best one. Greedy algorithms generate a single candidate: they don't explore alternatives. They can't be used when the problem asks for several best solution, e.g. if we wanted to know all largest subsets of non-overlapping intervals.

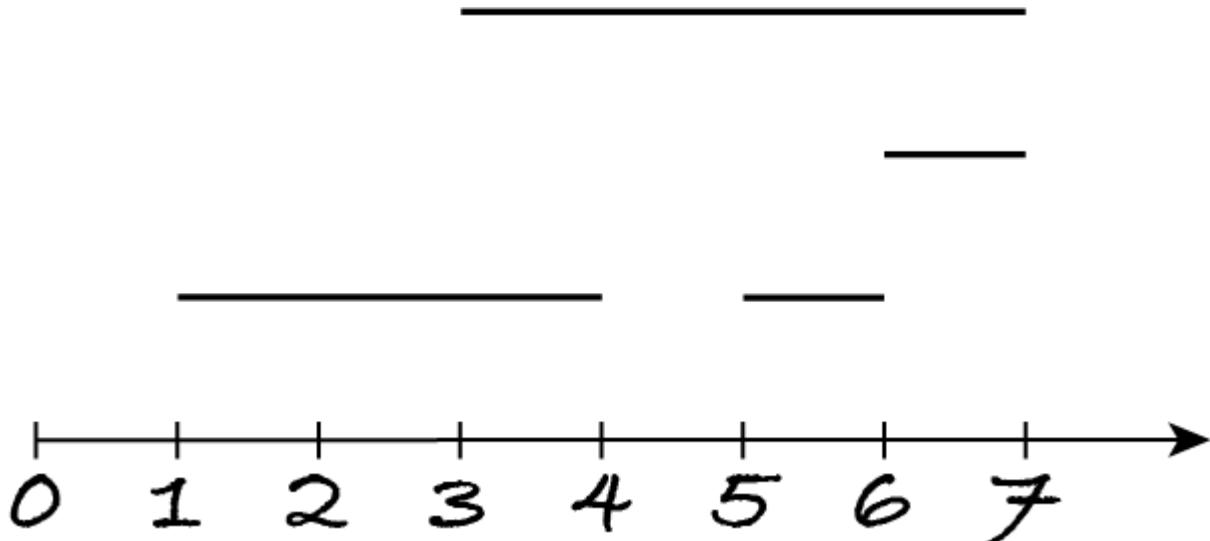
Exhaustive search generates in each iteration a complete candidate and tests it. Greedy algorithms extend a partial candidate by one item per iteration and they don't test the candidate at the end. If the greedy choice is correct, the generated candidate is a best solution.

18.1.2 Greedy choices

Let's now think about the greedy choice for the interval scheduling problem.

We want to select as many intervals as possible. One possible choice is to select the interval that starts earliest, so that we're not idle. Consider the intervals in the following figure.

Figure 18.1.1



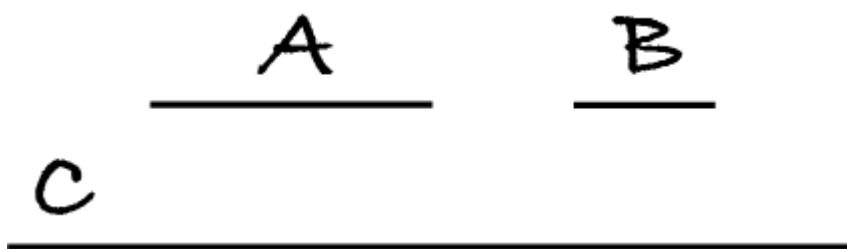
Processing them by ascending start time, i.e. from left to right in the figure, the algorithm:

1. adds (1, 4) to the empty schedule
2. skips (3, 7) because it overlaps with (1, 4)
3. adds (5, 6) to the schedule because it doesn't overlap (1, 4)
4. skips (6, 7) because it overlaps (5, 6) at time point 6.

The algorithm produces the schedule $\{(1,4), (5,6)\}$; it's a correct solution. No schedule with three intervals is possible.

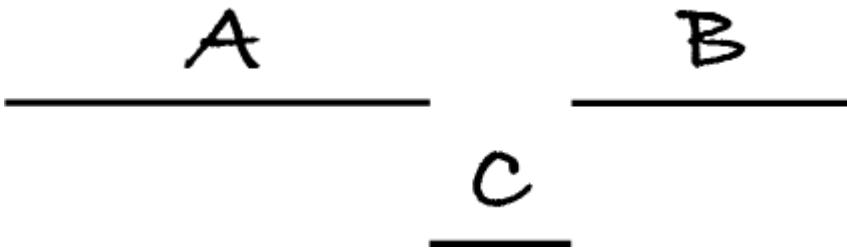
Unfortunately, choosing the earliest start doesn't always lead to a solution, as the next figure shows. The long interval C starts first but choosing it prevents finding the largest schedule, with intervals A and B.

Figure 18.1.2



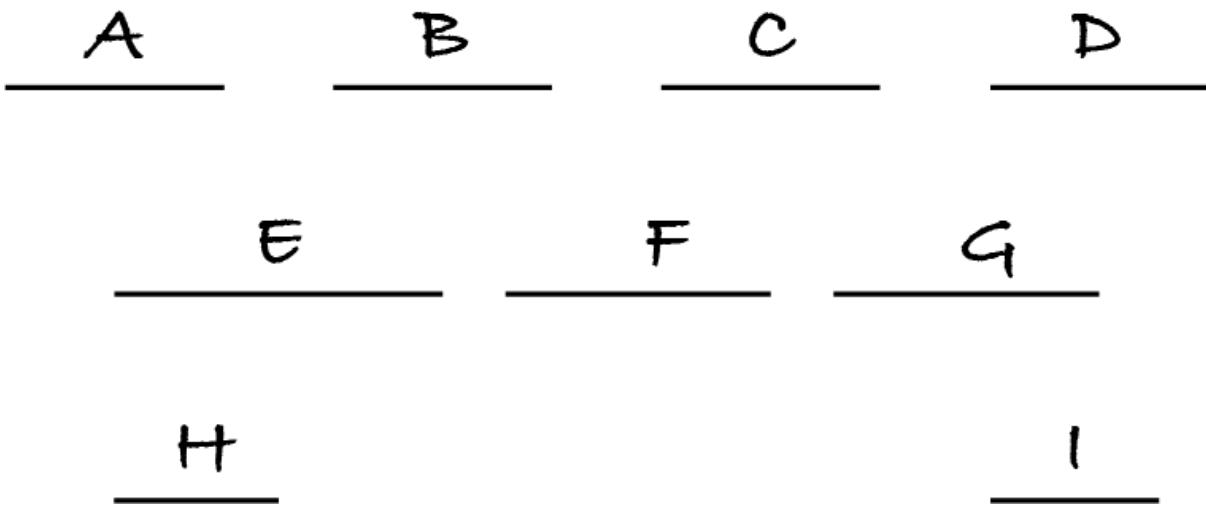
Well, if a single long interval prevents choosing several smaller intervals, let's choose as the best interval the shortest one. Alas, this isn't correct either, as the next figure shows. The shortest interval, C, overlaps intervals A and B at two time points: C starts when A ends and C ends when B starts. Therefore, choosing C first prevents finding the largest schedule $\{A, B\}$.

Figure 18.1.3



These counter-examples suggest we should choose the interval with the fewest overlaps, to remove the fewest other intervals. This sounds reasonable, but there's a counter-example for that. Consider the nine intervals in the following figure.

Figure 18.1.4

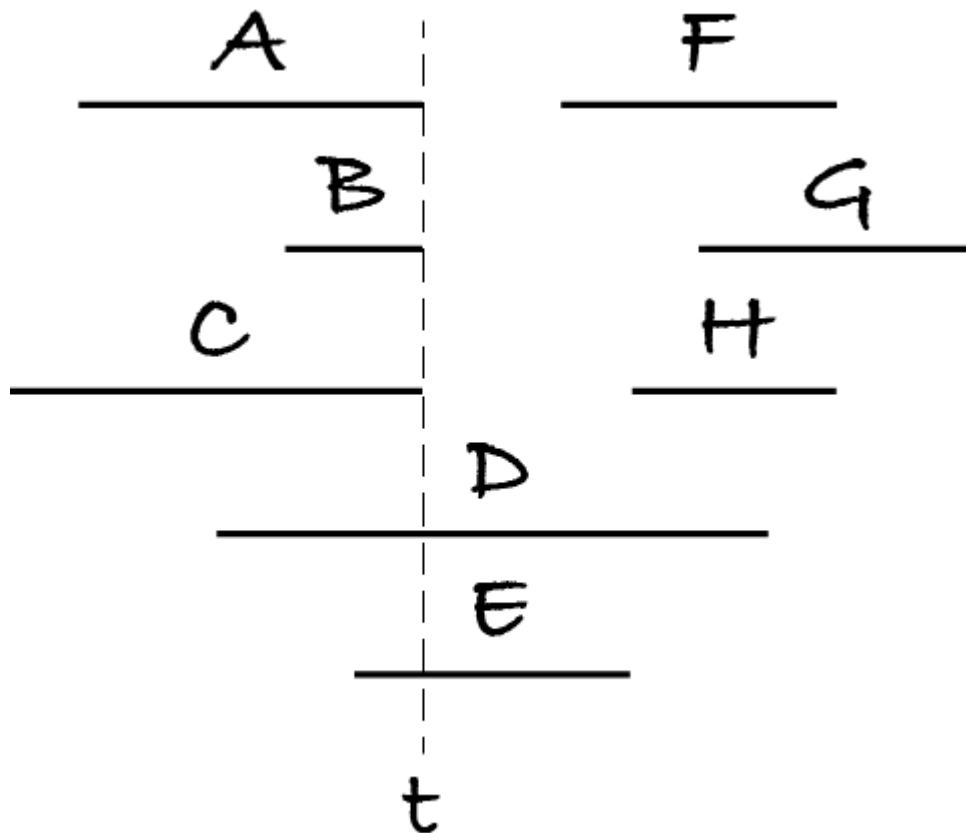


All intervals except E and G overlap two other intervals. The algorithm could choose any of them first, but if it chooses F, which eliminates B and C, the algorithm will only find a three-interval subset like {A, F, I} and not a four-interval solution like {A, B, C, D}.

Another possible choice is to pick the interval ending soonest, as that maximises the remaining time, allowing more intervals to be picked. Sorting the nine intervals by end time we get sequence (A, H, E, B, F, C, G, I, D). The algorithm adds A, skips H and E because they overlap A, adds B and thus skips F, adds C and then skips G, and finally adds I which forces D to be skipped. The output schedule is {A, B, C, I}, which satisfies the postconditions.

Choosing the interval ending soonest always leads to a largest subset. To understand why, consider the following diagram where t is the lowest ending time of all intervals.

Figure 18.1.5



Each interval belongs to one of three groups:

- it ends at t , like intervals A, B, and C
- it starts before t and ends after t , like intervals D and E
- it starts after t , like intervals F, G and H.

All intervals in the first two groups overlap each other, so the largest schedule has at most one of them, no matter how the intervals are chosen. If the algorithm chooses an interval in the second group, it may overlap with an interval in the third group, e.g. E overlaps with F, whereas the first and third groups never overlap. Thus, choosing any of the intervals ending soonest, at t , doesn't lead to fewer intervals in the schedule than any other choice, which means it's a largest schedule.



Note: Usually, there are several greedy choices for a problem, but possibly only one (or even none) will lead to a correct solution.

Proving that a greedy algorithm is correct can be hard; you won't be asked to do it. Proving that it's incorrect is usually relatively easy because a single counter-example will do. In his book *The Algorithm Design Manual*, Steven Skiena suggests four approaches for coming up with counter-examples.

- Think small: Algorithms fail under very particular conditions, which can be illustrated with small problem instances.

- Think exhaustively: There are only three ways to combine intervals (overlap, containment, disjoint); they can be used to construct examples systematically, e.g. one interval overlaps and thereby eliminates two other intervals.
- Go for a tie: Greedy algorithms choose the best extension, but if there are several best ones, the algorithm may choose the wrong one.
- Seek extremes: Counter-examples sometimes include items with opposite properties, like small and large, or long and short. For example, a long interval prevents choosing several small ones.



Note: Try constructing counter-examples that are small, contain items that are tied for being the best, or contain extreme items.

A counter-example reveals subtle conditions that make one greedy choice lead to an incorrect output. Thus they're good tests to check other greedy choices. For example, I checked the correct choice (interval ending soonest) on the counter-example for choosing the interval with fewest overlaps. You can check that the correct greedy choice also handles the counter-examples for choosing the interval starting soonest and for choosing the shortest interval.



Note: As you think of possible greedy choices, add counter-examples to a test table and check each greedy choice against them.

Exercise 18.1.1

In the introduction, I gave a greedy choice for the 0/1 knapsack problem: at each step take the most valuable remaining item that fits the knapsack. To do that we can initially sort the items by descending value. I also showed a counter-example for this choice.

Here are two other greedy choices:

1. Take the lightest item to keep space for more items, i.e. sort by ascending weight.
2. Take the most profitable item, i.e. sort by descending value-to-weight ratio.

For each greedy choice, provide a counter-example: a set of items and a knapsack capacity for which the greedy choice won't maximise the value of the items that can be carried.

Answer

18.1.3 Algorithm

Now that we have a greedy choice that leads to a largest schedule, we can apply the second greedy algorithmic pattern to interval scheduling, because the possible extensions are the intervals given in the input.

1. let *schedule* be the empty set

2. let *extensions* be *intervals* in ascending end time
3. for each *interval* in *extensions*:
 1. if *interval* is compatible with *schedule*:
 1. add *interval* to *schedule*

One way to implement step 3.1 is to check if the extension interval doesn't overlap any already scheduled interval. That takes linear time in the size of the partial schedule. We should try to exploit the sorted order for a more efficient approach.

When explaining why the algorithm is correct, I noted that only the intervals starting after the soonest ending time t are compatible with the intervals ending at t . So, if we keep the end time of the latest interval scheduled so far, we can easily check if the extension interval being considered is compatible.

1. let *schedule* be the empty set
2. let *extensions* be the *intervals* in ascending end time
3. let *last* be -1
4. for each *interval* in *extensions*:
 1. if the start of *interval* $>$ *last*:
 1. add *interval* to *schedule*
 2. let *last* be the end of *interval*

What's a worst-case scenario for this algorithm?

The algorithm does most work when no pair of intervals overlaps. In that scenario, the algorithm adds all intervals to the schedule.

Exercise 18.1.2

What's the worst-case complexity of the algorithm? Use $i = |\text{intervals}|$ for the size of the input.

Answer

Exercise 18.1.3 (optional)

Implement and test the algorithm. Write a test table with the above problem instances and counter-examples.

Exercise 18.1.4 (optional)

To appreciate how efficient the greedy approach is, outline an exhaustive search algorithm and explain its worst-case complexity.

18.2 Weighted graphs

The next examples of greedy algorithms will be on a new kind of graph, which this section introduces.

A graph shows which pairs of entities are related. Often we want to quantify the relationships, e.g. how long it takes or how much it costs to travel from A to B.

In a **weighted** graph, each edge has an associated number: its **weight**. The weight can represent distance, cost, duration, or anything else that's relevant for the model. For example, a weighted social network could represent the strengths of friendships. This is not asked of users, as it's unreliable and can be awkward, e.g. if Alice rates her friendship with Bob as 9 but Bob only rates it as 7. I imagine social media companies have their own secret friendship strength measure, e.g. based on mutual likes and posts, and use it to tailor recommendations, order the news feed, and in other features.

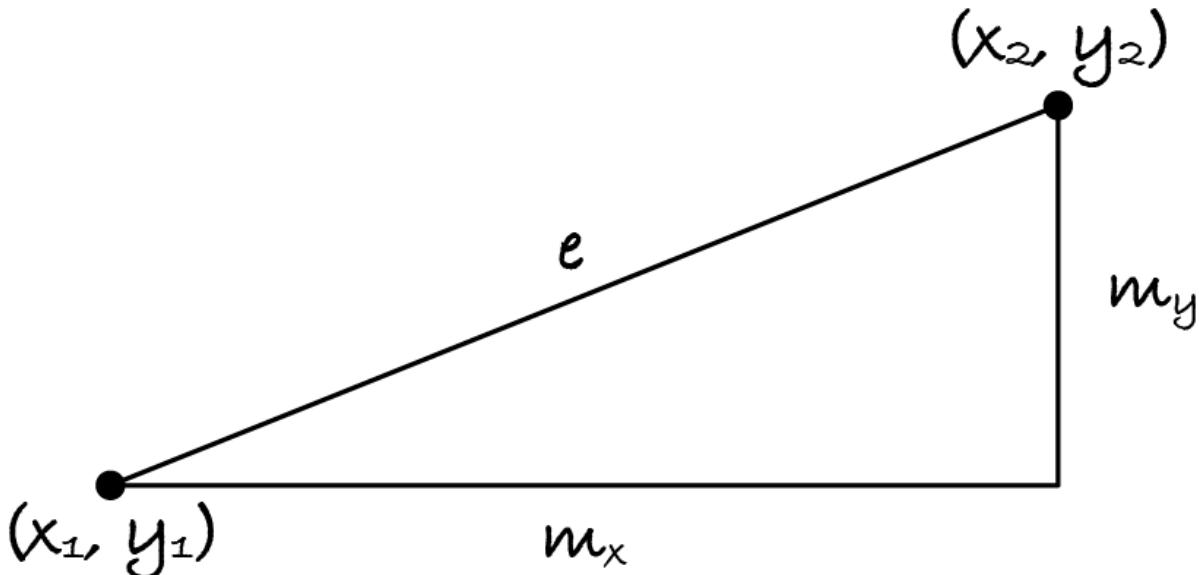
Edge weights can be any floating point number or integer, but examples will use mostly small integers. Depending on the problem and algorithm, there may be some restrictions, e.g. the weights must be positive or within a certain range.

For unweighted graphs, a shortest path between two nodes is a path with the fewest edges. For weighted graphs, a **shortest path** has the lowest sum of edge weights. For some reason, the term 'lightest path' hasn't caught on ...

Sometimes the weights aren't given in the problem and must be computed instead. For example, if nodes represent x - y points in a 2D space, the weights are usually some form of distance between them.

In the next figure, the **Manhattan distance** ('as the taxi drives') between two points is $m = m_x + m_y$ and the **Euclidean distance** ('as the crow flies') is obtained through the Pythagorean theorem: $e = \sqrt{m_x^2 + m_y^2}$.

Figure 18.2.1



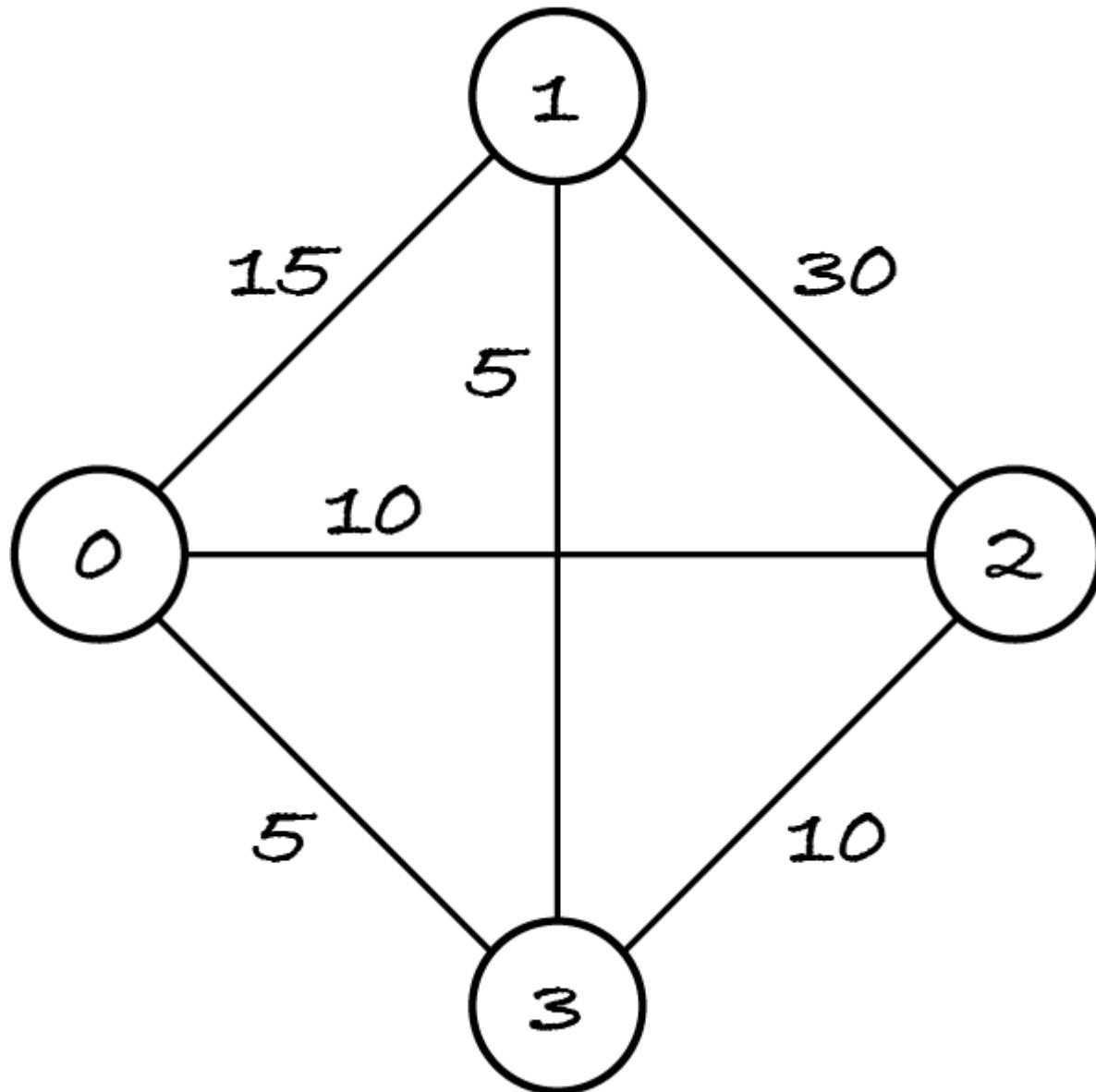
More generally, for any two points (x_1, y_1) and (x_2, y_2) represented with Python tuples:

- the Manhattan distance is `abs(x1 - x2) + abs(y1 - y2)`
- the Euclidean distance is `math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)`.

The Python function `abs` returns the absolute value of an expression: if the value is negative, it returns its negation, otherwise it returns the value. For example, `abs(-5)` and `abs(5)` both return 5. The absolute value is necessary for the Manhattan distance from A to B to be the same as from B to A.

Weighted graphs can be directed or undirected. You've seen the undirected weighted graph below when I introduced the *travelling salesman problem* (TSP): given a set of places and the costs of travelling between any two places, find a lowest-cost tour that visits all places once and returns to the start. Remember that for optimisation problems 'cost' is a generic term for the quantity being optimised: for the TSP it could be travel time or distance.

Figure 18.2.2



Now that you know about some *basic graph concepts*, we can define the TSP as a graph problem. The TSP is the optimisation problem of finding a shortest path that is a tour for a given

undirected weighted complete graph, where a tour is a cycle that includes all nodes.

18.2.1 Data structures

To store a weighted graph we have the same choices as for unweighted graphs.

To use an *edge list representation*, we represent edges as triples (*node1*, *node2*, *weight*) instead of pairs.

To use an *adjacency matrix representation*, we replace the Booleans with integers or floats indicating the weights. If there's no edge from A to B, we set the number in row A and column B to *infinity*. Here's the table for the graph above.

```
from math import inf

[
    [inf, 15, 10, 5],    # distances from node 0
    [15, inf, 30, 5],    # distances from node 1
    [10, 30, inf, 10],   # distances from node 2
    [5, 5, 10, inf]     # distances from node 3
]
```

When solving the TSP with an exhaustive search of all possible tours, I represented the input graph as an adjacency matrix, because it's complete, and filled the diagonal with zeros to indicate there's no cost in travelling from a place to itself. The diagonal is never used for the TSP, so it doesn't matter what values it has, but in general we must distinguish zero-weight edges from absent edges and therefore use infinity for the latter.

To use an *adjacency list representation*, we must add information about the weights of the edges going to the out-neighbours. There are several ways of doing so. Let's start with our representation of the unweighted version of the graph above.

```
{
    0: {1, 2, 3},
    1: {0, 2, 3},
    2: {0, 1, 3},
    3: {0, 1, 2}
}
```

For the weighted graph, we could represent neighbours with pairs (*node*, *weight*).

```
{
    0: {(1, 15), (2, 10), (3, 5)},
    1: {(0, 15), (2, 30), (3, 5)},
    2: {(0, 10), (1, 30), (3, 10)},
    3: {(0, 5), (1, 5), (2, 10)}
}
```

The representation states that there's an edge from node 0 to node 1 with weight 15, to node 2 with weight 10, to node 3 with weight 5, and similarly for the other nodes.

Checking if there's an edge from A to B is done in constant time for the unweighted graph, because B can be looked up directly in the set of A's neighbours. For the weighted graph, we must do a linear search over the pairs.

If we represent the neighbours as a dictionary of nodes to the weights of the edges, we can still check for an edge (A, B) in constant time: is B a key in the dictionary associated with A?

```
{  
    0: {1: 15, 2: 10, 3: 5},  
    1: {0: 15, 2: 30, 3: 5},  
    2: {0: 10, 1: 30, 3: 10},  
    3: {0: 5, 1: 5, 2: 10}  
}
```

I haven't done any space-time tradeoff. The second representation is better in terms of run-time *and* memory. Python implements sets and dictionaries with hash tables. The first representation uses a hash table where the keys are node-integer pairs and values are any object. The second representation uses a hash table where the keys are nodes and the values are integers. It uses fewer objects, and thus less space, than the first.

18.2.2 Classes

A weighted graph ADT has the same operations as for an unweighted graph, but the 'add edge' operation has an extra parameter for the weight, and there's a new operation to return the weight of an edge.

Again, I implement two classes, for directed and undirected graphs, the latter being a subclass of the former. The class for weighted digraphs is a subclass of its *unweighted counterpart*, so that I can inherit methods that are unaffected by the change of data structure.

```
[1]: %run -i ../m269_digraph  
%run -i ../m269_ungraph
```

```
[2]: # this code is also in m269_digraph.py  
  
import math  
  
  
class WeightedDiGraph(DiGraph):  
    """A weighted directed graph with hashable node objects.  
  
    Edges are between different nodes.  
    There's at most one edge from one node to another.  
    Edges have weights, which can be floats or integers.  
    """  
  
    def add_node(self, node: Hashable) -> None:  
        """Add the node to the graph.
```

(continues on next page)

(continued from previous page)

```

Preconditions: not self.has_node(node)
"""
self.out[node] = dict() # a map of out-neighbours to weights

def add_edge(self, start: Hashable, end: Hashable, weight:  
→float) → None:
    """Add edge start -> end, with the given weight, to the  
→graph.

    If the edge already exists, set its weight.

Preconditions:
self.has_node(start) and self.has_node(end) and start != end
"""
self.out[start][end] = weight

def weight(self, start: Hashable, end: Hashable) → float:
    """Return the weight of edge start -> end or infinity if it  
→doesn't exist.

Preconditions: self.has_node(start) and self.has_node(end)
"""
if self.has_edge(start, end):
    return self.out[start][end]
else:
    return math.inf

def remove_edge(self, start: Hashable, end: Hashable) → None:
    """Remove edge start -> end from the graph.

    If the edge doesn't exist, do nothing.

Preconditions: self.has_node(start) and self.has_node(end)
"""
if self.has_edge(start, end):
    self.out[start].pop(end)

def edges(self) → set:
    """Return the graph's edges as a set of triples (start, end,  
→weight)."""
    all_edges = set()
    for start in self.out:
        for (end, weight) in self.out[start].items():
            all_edges.add((start, end, weight))
    return all_edges

```

(continues on next page)

(continued from previous page)

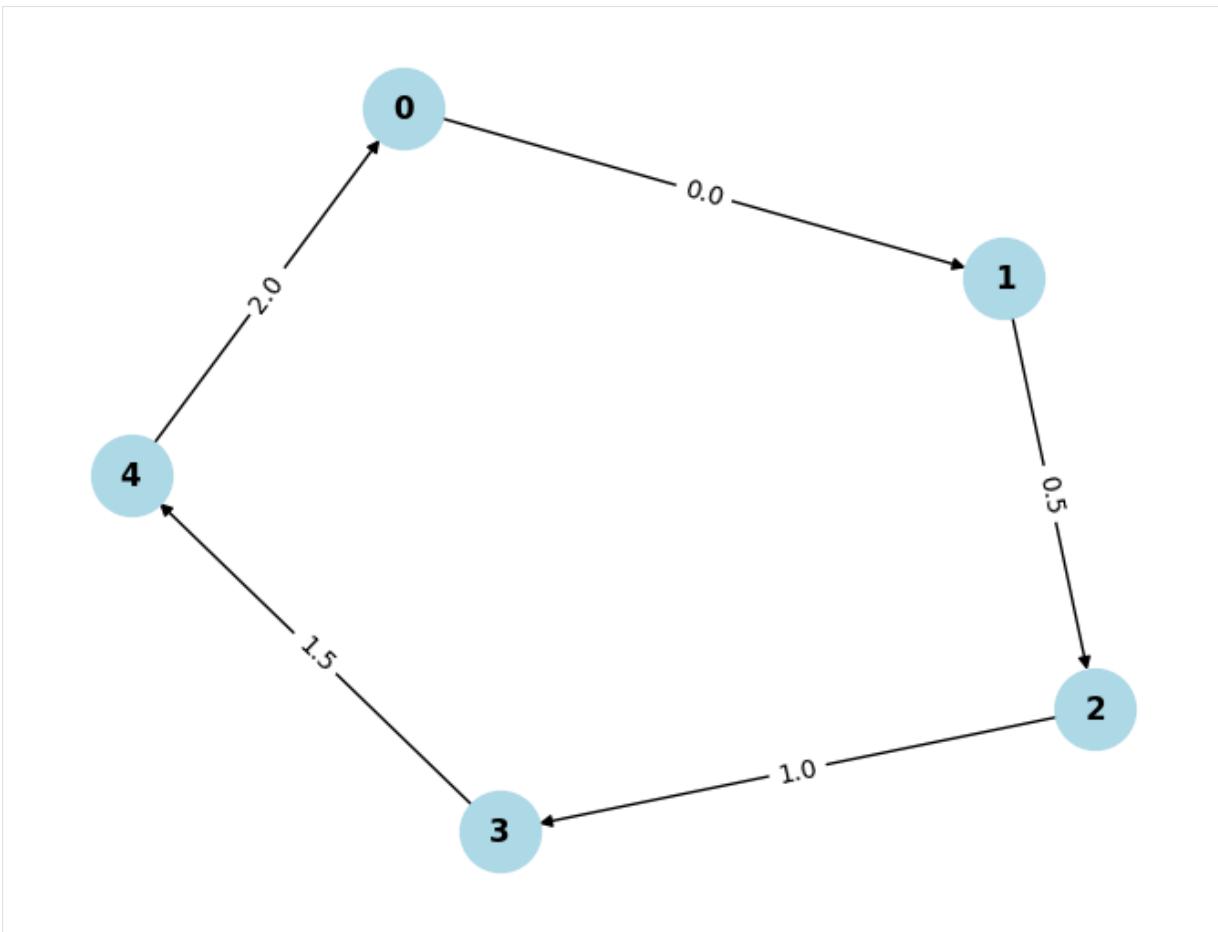
```
def out_neighbours(self, node: Hashable) -> set:
    """Return the out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return set(self.out[node].keys())

def draw(self) -> None:
    """Draw the graph."""
    if type(self) is WeightedDiGraph:
        graph = networkx.DiGraph()
    else:
        graph = networkx.Graph()
    graph.add_nodes_from(self.nodes())
    for (node1, node2, weight) in self.edges():
        graph.add_edge(node1, node2, w=weight)
    pos = networkx.spring_layout(graph)
    networkx.draw(
        graph,
        pos,
        with_labels=True,
        node_size=1000,
        node_color="lightblue",
        font_size=12,
        font_weight="bold",
    )
    networkx.draw_networkx_edge_labels(
        graph, pos, edge_labels=networkx.get_edge_attributes(graph, "w")
    )
```

Let's create an example graph to see how weights are depicted.

```
[3]: graph = WeightedDiGraph()
for node in range(5):
    graph.add_node(node)
for node in range(5):
    graph.add_edge(node, (node + 1) % 5, node / 2)
graph.draw()
```



Now the class for weighted undirected graphs, which is very similar to its *unweighted counterpart*: each undirected weighted edge is represented as two directed weighted edges, and there's no distinction between in-degree, out-degree and degree, and likewise for neighbours.

[4]: # this code is also in m269_ungraph.py

```

class WeightedUndirectedGraph(WeightedDiGraph):
    """A weighted undirected graph with hashable node objects.

    There's at most one edge between two different nodes.
    There are no edges between a node and itself.
    Edges have weights, which may be integers or floats.
    """

    def add_edge(self, node1: Hashable, node2: Hashable, weight:_float) -> None:
        """Add an edge node1-node2 with the given weight to the graph.

        If the edge already exists, set its weight.
        """
  
```

(continues on next page)

(continued from previous page)

```

Preconditions: self.has_node(node1) and self.has_node(node2)
"""

super().add_edge(node1, node2, weight)
super().add_edge(node2, node1, weight)

def remove_edge(self, node1: Hashable, node2: Hashable) -> None:
    """Remove edge node1-node2 from the graph.

    If the edge doesn't exist, do nothing.

Preconditions: self.has_node(node1) and self.has_node(node2)
"""

super().remove_edge(node1, node2)
super().remove_edge(node2, node1)

def edges(self) -> set:
    """Return the graph's edges as a set of triples (node1,_
    →node2, weight).

Postconditions: for every edge A-B,
the output has either (A, B, w) or (B, A, w) but not both
"""
    all_edges = set()
    for start in self.out:
        for (end, weight) in self.out[start].items():
            if (end, start, weight) not in all_edges:
                all_edges.add((start, end, weight))
    return all_edges

def in_neighbours(self, node: Hashable) -> set:
    """Return all nodes that are adjacent to the node.

Preconditions: self.has_node(node)
"""

return self.out_neighbours(node)

def neighbours(self, node: Hashable) -> set:
    """Return all nodes that are adjacent to the node.

Preconditions: self.has_node(node)
"""

return self.out_neighbours(node)

def in_degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

```

(continues on next page)

(continued from previous page)

```

Preconditions: self.has_node(node)
"""
return self.out_degree(node)

def degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

Preconditions: self.has_node(node)
"""
return self.out_degree(node)

```

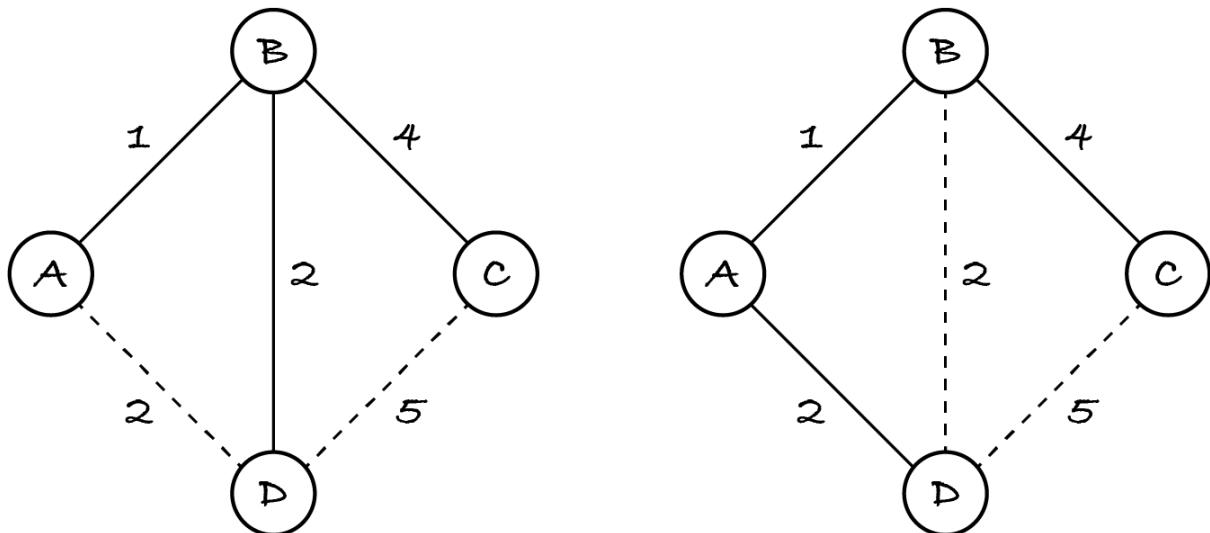
18.3 Minimum spanning tree

A city council wants to control all traffic lights at junctions in real-time, from a control centre, so that traffic flow can be improved. For that, they need to lay fibre optic cable under the streets, from each junction to the control centre. The council wants to reduce costs by laying as little cable as possible. Where should cable be laid?

We can model the city as an undirected weighted graph with one node for the centre, one node per junction and one edge per street. The weights represent the lengths of the streets. The problem requires selecting a subset of the edges, with the least total weight, so that all junction nodes are connected to the centre node.

The next figure shows two solutions, with total weight 7, for the same graph. The dashed lines are the graph edges not included in the solution. Node C represents the control centre.

Figure 18.3.1



A **spanning tree** of an undirected graph G is a subgraph of G that is a tree, i.e. is connected and acyclic, and has all nodes of G . The tree spans over all nodes, hence its name. A spanning tree is a subgraph with the fewest edges that still connect all nodes. If G isn't connected, then it has no spanning tree; otherwise it may have multiple spanning trees.

If G is weighted, then a **minimum spanning tree** (MST) of G is a spanning tree with the lowest possible sum of weights.

The problem given at the start is thus a particular formulation of the more general problem of computing an MST of a weighted undirected connected graph. Finding an MST solves the problem of connecting all nodes in the shortest, cheapest or fastest way, depending on what the weights represent.

We'll look at two versions of the same greedy algorithm to construct an MST.

18.3.1 First algorithm

The following algorithm constructs a tree incrementally, one node at a time, from a given *start* node. At each step, it adds an edge between a node that is in the tree and one that isn't. This avoids introducing cycles, to guarantee that the subgraph constructed is a tree.

Each step adds one more node to the tree, so to obtain a spanning tree we stop after $n - 1$ iterations, where n is the number of nodes. To obtain a minimum spanning tree, we choose at each step the edge with the lowest weight. This will lead to the least total weight. I'll explain why later. First the algorithm.

1. let $tree$ be the weighted undirected graph with single node *start*
2. repeat $n - 1$ times:
 1. find an edge (A, B) with lowest weight(A, B) such that A is in $tree$ but B isn't
 2. add node B to $tree$
 3. add edge $(A, B, \text{weight}(A, B))$ to $tree$

At each step, the algorithm's greedy choice is to add the node 'nearest' to the tree. For the example above, starting at node C, the algorithm first adds the edge to node B, then to node A and finally to node D. It constructs the MST on the right, not the one on the left.

If the algorithm starts at node D, then the first iteration may add the edge to B or the edge to A and thus lead to the left or right MST, respectively.

This algorithm is commonly known as **Prim's algorithm**.



Info: The algorithm was first invented and published in 1930 by Vojtěch Jarník and independently reinvented by Robert Prim in 1957 and Edsger Dijkstra in 1959.

Step 2.1 can be done with a linear search over all edges. The algorithm has complexity $(n - 1) \times \Theta(e) = \Theta(n \times e - e) = \Theta(n \times e)$.

Does it help to first sort the edges by ascending weight?

Sorting the edges helps to make the search faster in practice, as it can stop when it finds the first edge between a tree node and a non-tree node. However, the complexity gets worse: $\Theta(e \log e) + \Theta(n \times e)$.

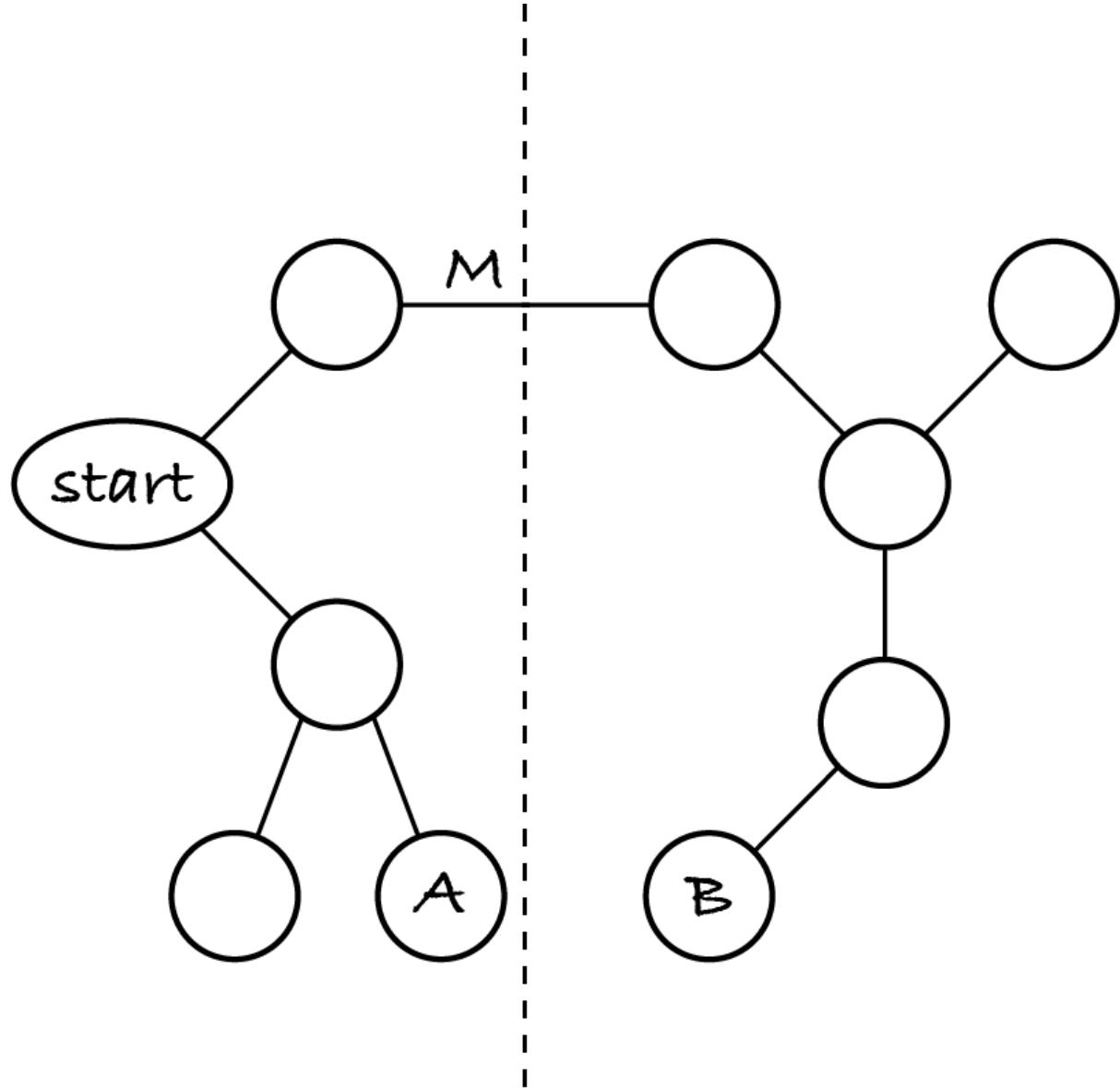
If the edges are sorted, why not use binary search instead of linear search to reduce the complexity to $\Theta(e \log e) + \Theta(n \times \log e) = \Theta((e + n) \log e)$?

We can't use binary search because we're looking for an edge between two nodes with specific characteristics. Let's suppose the edge in the middle is between two tree nodes or between two non-tree nodes. That doesn't tell us whether an edge between a tree node and a non-tree node is to the left or to the right.

This simple version of Prim's algorithm is inefficient because it searches all edges over and over although at any point in time there are relatively few edges between nodes in the tree and nodes outside the tree.

Even though the algorithm is inefficient, it's correct. At each step, it chooses an edge that does belong to the MST. To see why, consider the following diagram of the MST of some graph. I've omitted the weights and most of the node labels.

Figure 18.3.2



The subtree to the left of the dashed line is the part of the MST the algorithm has constructed so far, beginning with the start node. The subtree to the right of the dashed line is the rest of the MST, with the yet-unvisited nodes. Edge M is the one edge in the MST that connects both subtrees.

Why is there exactly one such edge? First, there must be at least one edge connecting both subtrees, otherwise the MST wouldn't span all nodes. Second, there cannot be two edges connecting both subtrees, because then the MST would have a cycle: we could go from the start node to the right subtree via one edge and return to the start via another. Since MSTs are trees, they don't have cycles.

Among all edges the input graph has between the left subtree and the right subtree, edge M has the lowest weight. To see why, imagine the graph has an edge between nodes A and B in the diagram. If its weight were lower than M's, we could replace edge M with edge (A, B) to obtain a tree that spans all nodes and has a lower total weight. That's contrary to the starting assumption that the tree with edge M is a minimum spanning tree. So M has the lowest weight among all edges connecting the left and right subtrees. Therefore Prim's algorithm will pick it.

If edge (A, B) has the same weight as edge M, then there are two MSTs: one has M and the other has (A, B), so it doesn't matter which one Prim's algorithm picks: both lead to a spanning tree with the same total weight.

Exercise 18.3.1

Inspired by Prim's algorithm, Bob invents a greedy algorithm to solve the TSP. Beginning with the start node, it extends the tour by one node at a time, adding the 'nearest' node to the currently last node in the tour. After doing this $n - 1$ times, it adds the edge from the last node to the start node. The input graph is complete, so that edge always exists and completes the tour.

Here's Prim's algorithm again. Modify it to become Bob's algorithm.

1. let *tree* be the weighted undirected graph with single node *start*
2. repeat $n - 1$ times:
 1. find an edge (A, B) with lowest weight(A, B) such that A is in *tree* but B isn't
 2. add node B to *tree*
 3. add edge (A, B, weight(A, B)) to *tree*

Hint Answer

Exercise 18.3.2

As Bob thinks of some tests for his algorithm, he realises it doesn't solve the TSP. Show a counter-example.

Hint Answer

Bob's greedy algorithm is an example of a *heuristic algorithm*: it doesn't compute the optimal solution, only an approximate one, but it's much faster than the correct brute-force search algorithm. The greedy algorithm has the same complexity as Prim's algorithm, $\Theta(n \times e)$, whereas searching all tours, i.e. all node permutations, for the shortest one takes $\Theta(n!)$.

18.3.2 Second algorithm

Prim's algorithm constructs a tree one edge at a time, to a node not in the tree. We already have a similar algorithm: *graph traversal*. It also produces a tree, visiting one new node at a time. Each iteration processes the next edge. If it connects two visited nodes, it's discarded; otherwise the node it leads to is visited and its out-going edges are added to those yet to be processed.

We get different traversals depending on the unprocessed edges collection. If it's a set, we get a random traversal. If it's a queue, we get a *breadth-first search* because the edges discovered first are followed before the others. If it's a stack, we get a depth-first search because the last discovered edge is followed next.

If we instead use a *min-priority queue*, where the priority is the edge weight, we get Prim's algorithm because the next edge followed is the one with lowest weight. Here is the generic traversal algorithm, with the necessary changes: the input and output graphs are weighted and undirected, and the edge weights are used as priorities. Remember that $\max(pq)$ represents the highest-priority item in priority queue pq .

1. let $visited$ be an undirected weighted graph with node $start$
2. let $unprocessed$ be an empty min-priority queue
3. for each $node$ in neighbours of $start$ in $graph$:
 1. add $(start, node)$ with priority $\text{weight}(start, node)$ to $unprocessed$
4. while $unprocessed$ isn't empty:
 1. let $(previous, current)$ be $\max(unprocessed)$ with priority $weight$
 2. remove $\max(unprocessed)$
 3. if $visited$ doesn't have node $current$:
 1. add $current$ to $visited$
 2. add $(previous, current, weight)$ to $visited$
 3. for each $node$ in neighbours of $current$ in $graph$:
 1. add $(current, node)$ with priority $\text{weight}(current, node)$ to $unprocessed$

Like the previous traversal algorithms, this one visits each node once, so steps 1 and 4.3.1 take in total $\Theta(n)$ time.

All edges of a just-visited node are added to the priority queue, so each edge (A, B) is added twice: first when visiting A, next when visiting B. If a priority queue is implemented with a heap, adding and removing an item takes logarithmic time in the length of the queue at that moment. The priority queue has at most $2 \times e$ edges (usually far fewer), so processing the edges takes $2 \times e \times O(\log 2 \times e) = O(e \log e)$.

The overall complexity of this version of Prim's algorithm is $\Theta(n) + O(e \log e) = O(e \log e)$. A tree has $n - 1$ edges, so connected graphs have $e \geq n - 1$ edges. Hence $e \log e \geq n$ and we can ignore the slower-growing term $\Theta(n)$.

Exercise 18.3.3

Suggest a change to the algorithm that, while not reducing its complexity, will reduce the run-time for many graphs.

Hint Answer

Exercise 18.3.4

Given an undirected weighted connected graph in which all weights are the same, can we compute an MST of that graph without using Prim's algorithm?

Hint Answer

18.3.3 Code

Let's implement and run the algorithm. I'll leave it as an exercise to implement the run-time improvements of Exercise 18.3.3.

```
[1]: %run -i ../m269_digraph  
%run -i ../m269_ungraph
```

I implement the priority queue with *Python's heap functions*. Since they don't separate the priority from the item, I must add edges to the heap as triples (*weight, node1, node2*) for the heap to sort them by weight. If two edges have the same weight, they will be sorted by the nodes, so we must assume that the node objects are comparable.

```
[2]: # this code is also in m269_ungraph.py  
  
from heapq import heappush, heappop  
  
  
def prim(graph: WeightedUndirectedGraph, start: Hashable) ->  
    WeightedUndirectedGraph:  
    """Return a minimum spanning tree of graph, beginning at start.  
  
    Preconditions:  
    - graph.has_node(start)  
    - graph is connected  
    - node objects are comparable  
    """  
    visited = WeightedUndirectedGraph()  
    visited.add_node(start)  
  
    unprocessed = []  
    for neighbour in graph.neighbours(start):  
        weight = graph.weight(start, neighbour)  
        heappush(unprocessed, (weight, start, neighbour))
```

(continues on next page)

(continued from previous page)

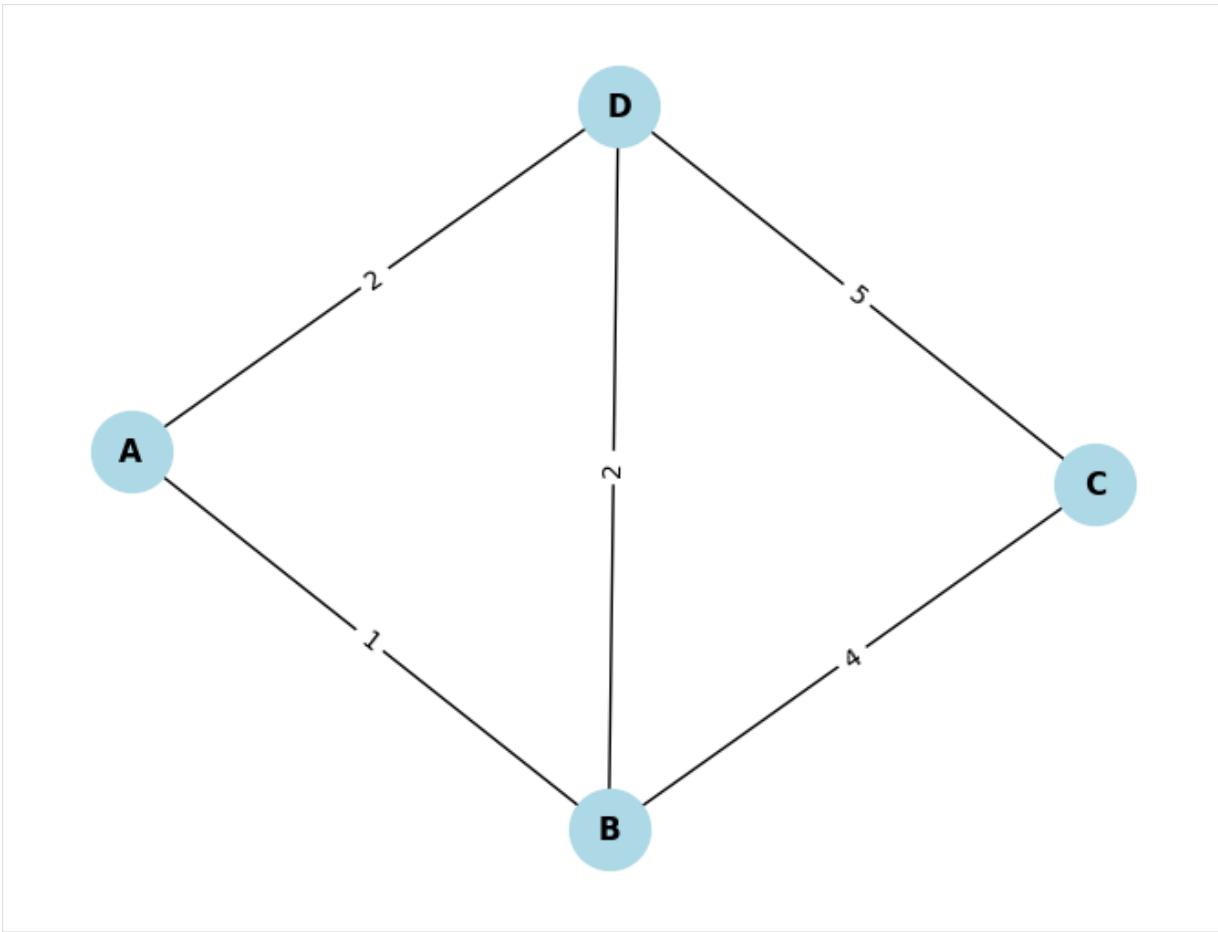
```
while len(unprocessed) > 0:
    edge = heappop(unprocessed)
    weight = edge[0]
    previous = edge[1]
    current = edge[2]
    if not visited.has_node(current):
        visited.add_node(current)
        visited.add_edge(previous, current, weight)
        for neighbour in graph.neighbours(current):
            weight = graph.weight(current, neighbour)
            heappush(unprocessed, (weight, current, neighbour))
return visited
```

Next I create the example graph.

```
[3]: # this code is also in m269_graphs.py
```

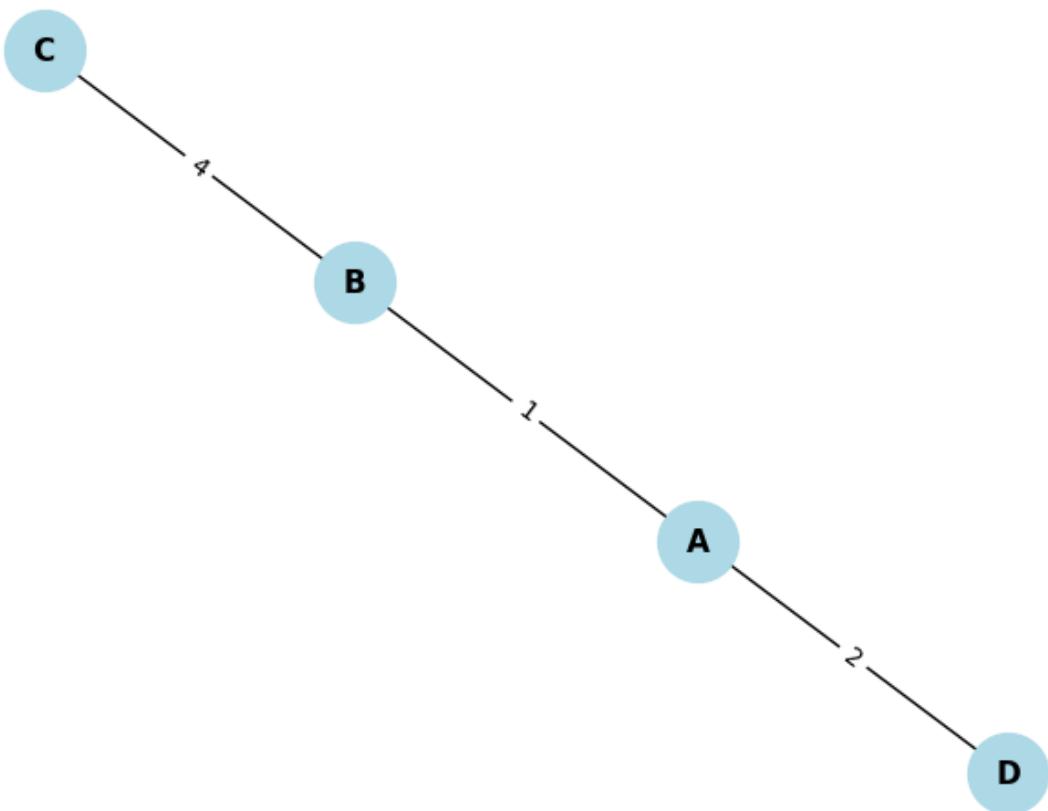
```
# graph in Figure 18.3.1
RHOMBUS = WeightedUndirectedGraph()
for node in "ABCD":
    RHOMBUS.add_node(node)
RHOMBUS.add_edge("A", "B", 1)
RHOMBUS.add_edge("A", "D", 2)
RHOMBUS.add_edge("B", "C", 4)
RHOMBUS.add_edge("B", "D", 2)
RHOMBUS.add_edge("C", "D", 5)
```

```
[4]: RHOMBUS.draw()
```



Edge (A, D) is always chosen before (B, D) because $(2, 'A', 'D') < (2, 'B', 'D')$ when the heap compares them. The function always produces the same MST, of the two possible, no matter what start node we select.

```
[5]: prim(RHOMBUS, "D").draw() # replace D with A, B or C
```



18.4 Shortest paths

This section introduces an algorithm to solve the **single-source shortest paths** (SSSP) problem for weighted graphs: for each node R that is reachable from the given start node S, find a lowest cost (total weight) path from S to R. A typical application of the SSSP problem is to find the cheapest or fastest way to go from one place to every other place in a transport network.

We defined the MST problem for connected undirected graphs to guarantee that a spanning tree can be obtained, no matter the start node. The SSSP problem is defined for any directed or undirected graph, because it doesn't ask to connect all nodes. However, the algorithm to be shown only works for graphs with non-negative weights; most models of real-world networks don't have negative weights, so this isn't much of a restriction in practice.

18.4.1 Algorithm

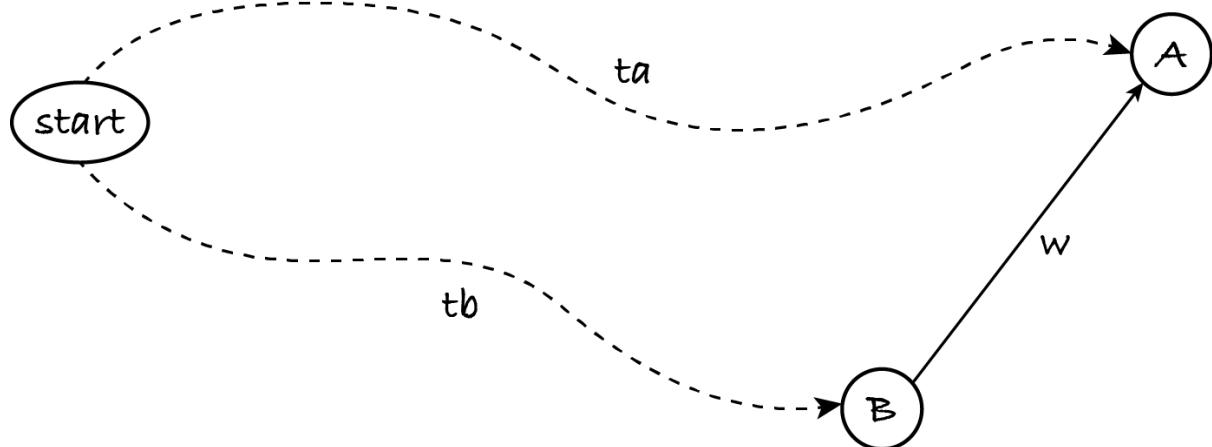
The algorithm is again a graph traversal that produces a tree rooted at the start node, with a single path to each reachable node. In fact, it's a small modification of Prim's algorithm.

Prim's algorithm greedily chooses in each iteration the unvisited node 'nearest' to *any* visited node, because it's minimising the total weight of the tree. The SSSP algorithm chooses instead the unvisited node 'nearest' to *one* visited node, namely the start node, because it's minimising the weight of the paths beginning at the start node. This is known as **Dijkstra's algorithm**, named after its inventor.

You should now watch a [visualisation](#) that explains the gist of Dijkstra's algorithm, applied to a connected undirected graph in which weights represent distances. The visualisation doesn't create a tree with the shortest paths: it only computes their weights. Extra work is then needed to obtain the paths.

Now that you've seen the algorithm in action, let's see why it works. The next figure shows schematically the shortest paths to nodes A and B, with total weights ta and tb , respectively. In addition there's an edge with non-negative weight w from B to A.

Figure 18.4.1



Let's assume that A is visited before B by the algorithm. This means that A is 'nearer' to the start node than B, i.e. $ta \leq tb$ and therefore $ta \leq tb + w$. In other words, the path to A via B can't be shorter. Therefore, as soon as the algorithm picks A to visit next, it knows it has found a shortest path. Paths via nodes visited later won't be shorter.

Note that if $w < 0$ then we may have $ta > tb + w$. If weights can be negative, the greedy choice of picking the next nearest node won't always work: a shorter path may go first to a node further off and then take a negative weight edge to obtain a lower total weight.

As I mentioned above, the visualisation doesn't construct a tree. So let's take our version of Prim's algorithm and use different priorities. My version of Prim's algorithm adds edge (A, B, w) with priority w to the min-priority queue of unprocessed edges. For Dijkstra's, the priority is the cost (total weight) of reaching B via A, i.e. it's the cost to reach A plus w .

How do I know the cost of reaching A? Well, the graph traversal only adds edge (A, B, w) when visiting A, and it visits A because it took from the queue some incoming edge (C, A) with priority p which is the cost of reaching A.

1. let *visited* be a weighted digraph with node *start*
2. let *unprocessed* be an empty min-priority queue
3. for each *node* in the out-neighbours of *start* in *graph*:
 1. add edge $(start, node, weight)$ with priority *weight* to *unprocessed*
4. while *unprocessed* isn't empty:
 1. let $(previous, current, weight)$ be $\max(unprocessed)$ with priority *cost*
 2. remove $\max(unprocessed)$

3. if *visited* hasn't got node *current*:
 1. add *current* to *visited*
 2. add (*previous*, *current*, *weight*) to *visited*
 3. for each *node* in the out-neighbours of *current* in *graph*:
 1. let *weight* be weight(*current*, *node*)
 2. add edge (*current*, *node*, *weight*) with priority *cost* + *weight* to *unprocessed*

The only changes to Prim's algorithm are to use out-neighbours instead of neighbours in steps 3 and 4.3.3, to accommodate digraphs, and computing a different priority in step 4.3.3.2.

The changes don't affect the worst-case complexity, which is $O(e \log e)$, like for Prim's algorithm.



Info: The worst-case complexity of both algorithms is often stated as $O(e \log n)$. Any graph has fewer than n^2 edges and $\log x^y = y \log x$, so $O(e \log e) = O(e \log n^2) = O(e \times 2 \times \log n) = O(e \log n)$.

My versions of BFS, DFS, Prim's and Dijkstra's algorithms aim to highlight their similarities: they use the same core graph-traversal algorithm, only differing in the order in which unprocessed edges are stored.

Exercise 18.4.1

If the weights in the input graph are all equal, can you compute the shortest paths without using Dijkstra's algorithm?

Hint Answer

18.4.2 Code

Let's implement and run the algorithm.

```
[1]: %run -i ./m269_digraph
%run -i ./m269_ungraph
%run -i ./m269_graphs
```

Like for Prim's algorithm, I use Python's min-heaps.

```
[2]: # this code is also in m269_digraph.py
```

```
from heapq import heappush, heappop
```

```
def dijkstra(graph: WeightedDiGraph, start: Hashable) -> _
```

(continues on next page)

(continued from previous page)

```

→WeightedDiGraph:
    """Return a shortest path from start to each reachable node.

Preconditions:
- graph.has_node(start)
- node objects are comparable
- no weight is negative
"""

visited = WeightedDiGraph()
visited.add_node(start)

# create min-priority queue of tuples (cost, (A, B, weight))
# cost is total weight from start to B via shortest path to A
unprocessed = [] # min-priority queue
for neighbour in graph.out_neighbours(start):
    weight = graph.weight(start, neighbour)
    heappush(unprocessed, (weight, (start, neighbour, weight)))

while len(unprocessed) > 0:
    info = heappop(unprocessed)
    cost = info[0]
    edge = info[1]
    previous = edge[0]
    current = edge[1]
    weight = edge[2]

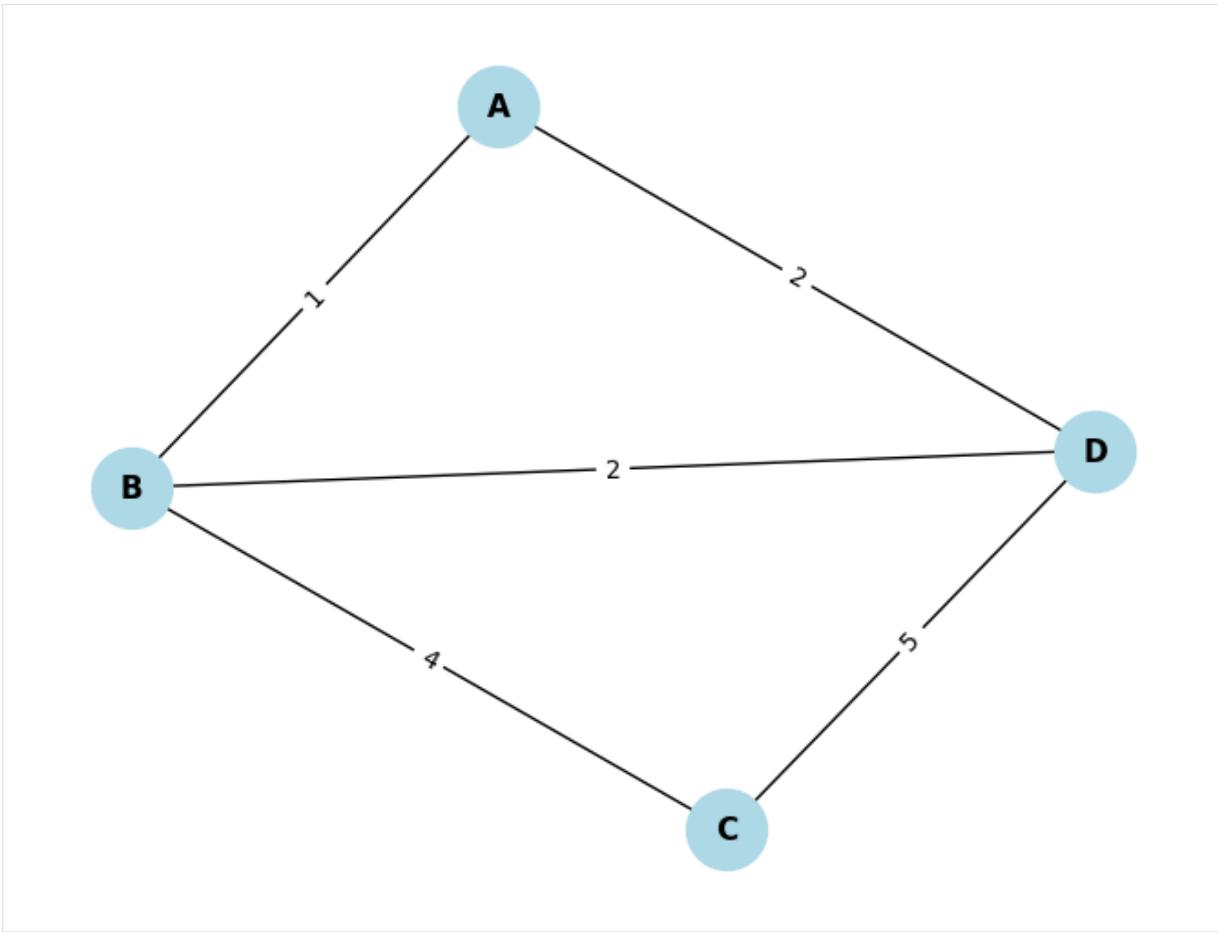
    if not visited.has_node(current):
        visited.add_node(current)
        visited.add_edge(previous, current, weight)
        for neighbour in graph.out_neighbours(current):
            weight = graph.weight(current, neighbour)
            edge = (current, neighbour, weight)
            heappush(unprocessed, (cost + weight, edge))

return visited

```

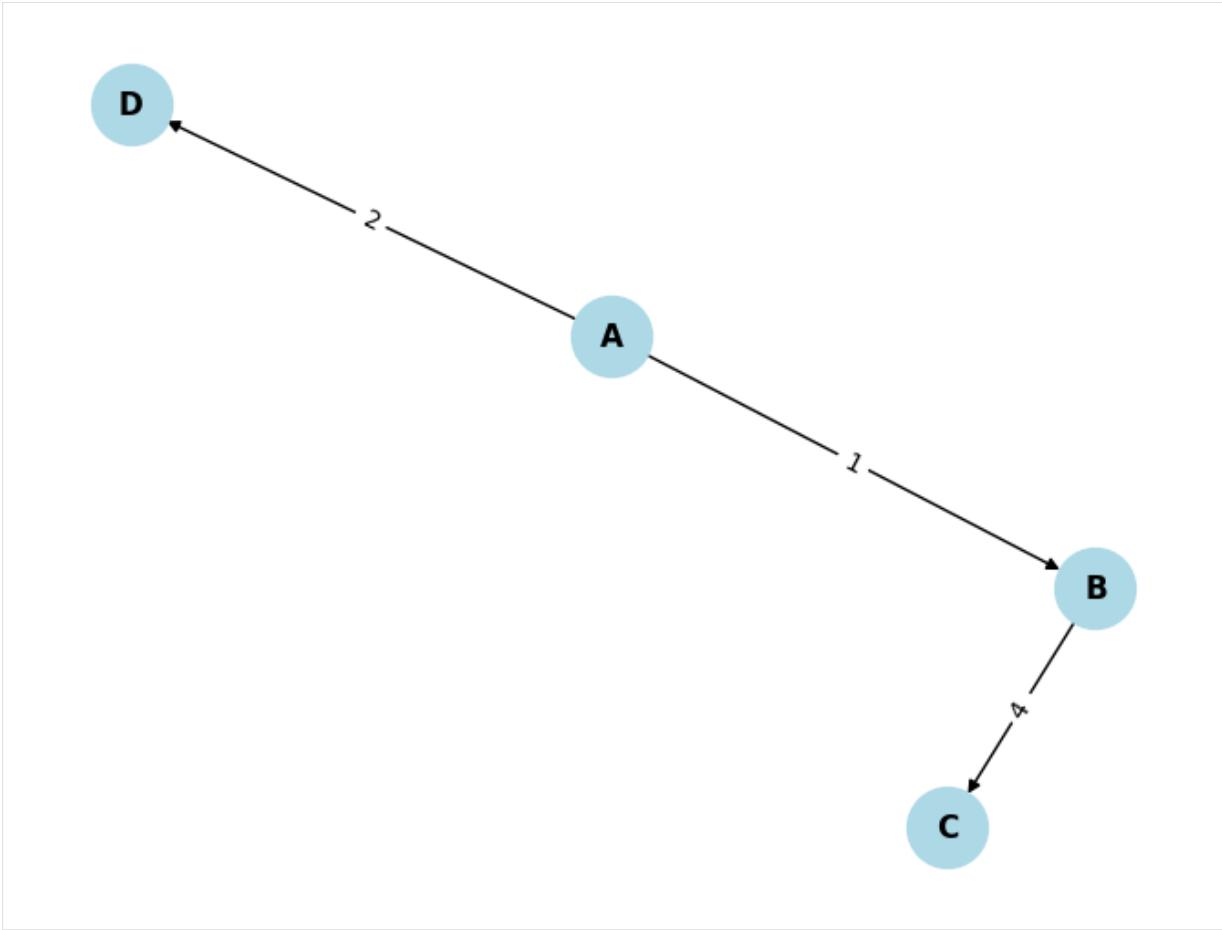
Let's use the example MST graph of [Section 18.3](#).

[3]: RHOMBUS.draw()



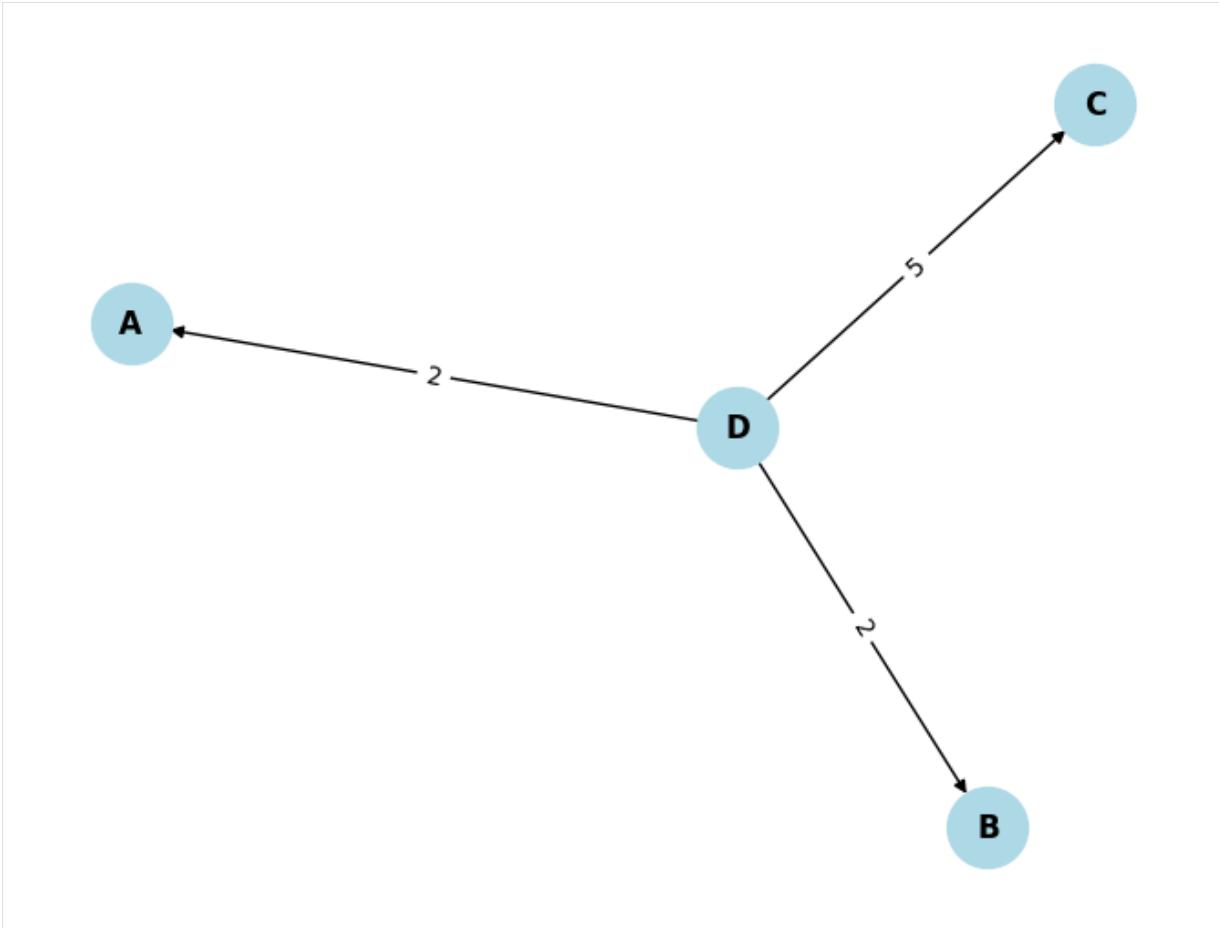
If the start node is A then the shortest paths happen to coincide with the MST, if we ignore the edge directions of the output digraph.

```
[4]: dijkstra(RHOMBUS, "A").draw()
```



However, if the start node is D then we get a different spanning tree.

```
[5]: dijkstra(RHOMBUS, "D").draw()
```



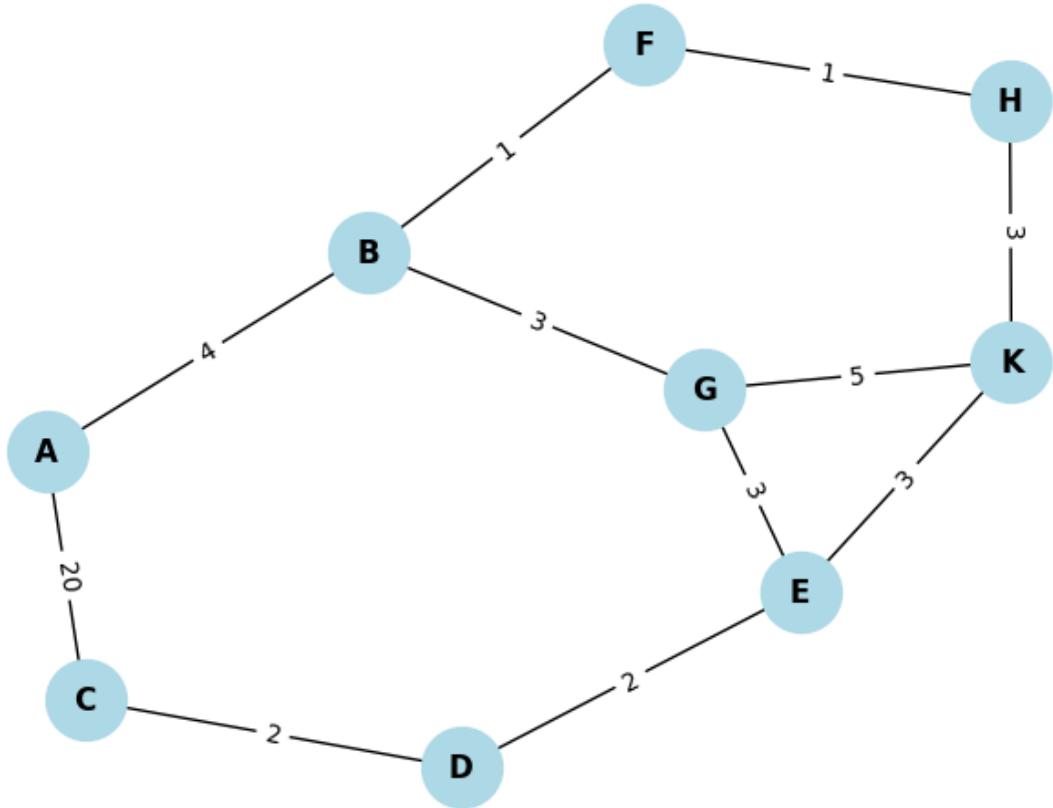
Finding the MST involves minimising the total weight of *all edges*. Finding shortest paths involves minimising the weight of *each path*: hence the output tree is often not an MST.

The interactive visualisation graph is another example of that.

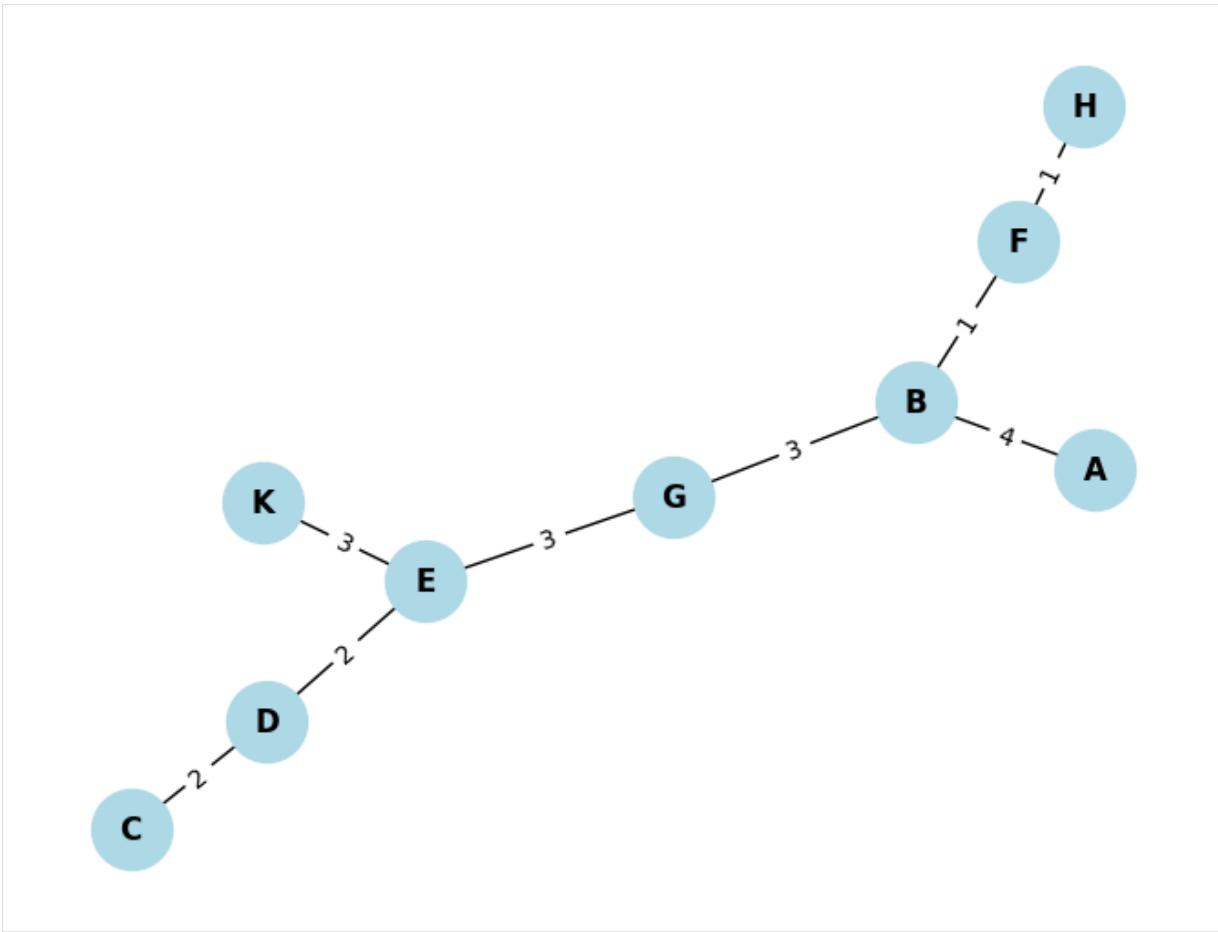
```
[6]: # this code is also in m269_graphs.py

# graph used by the interactive visualisation of Dijkstra's algorithm
DIJKSTRA = WeightedUndirectedGraph()
for node in "ABCDEFGHIK":
    DIJKSTRA.add_node(node)
DIJKSTRA.add_edge("C", "A", 20)
DIJKSTRA.add_edge("A", "B", 4)
DIJKSTRA.add_edge("B", "F", 1)
DIJKSTRA.add_edge("F", "H", 1)
DIJKSTRA.add_edge("B", "G", 3)
DIJKSTRA.add_edge("G", "K", 5)
DIJKSTRA.add_edge("K", "H", 3)
DIJKSTRA.add_edge("C", "D", 2)
DIJKSTRA.add_edge("D", "E", 2)
DIJKSTRA.add_edge("E", "G", 3)
DIJKSTRA.add_edge("E", "K", 3)
```

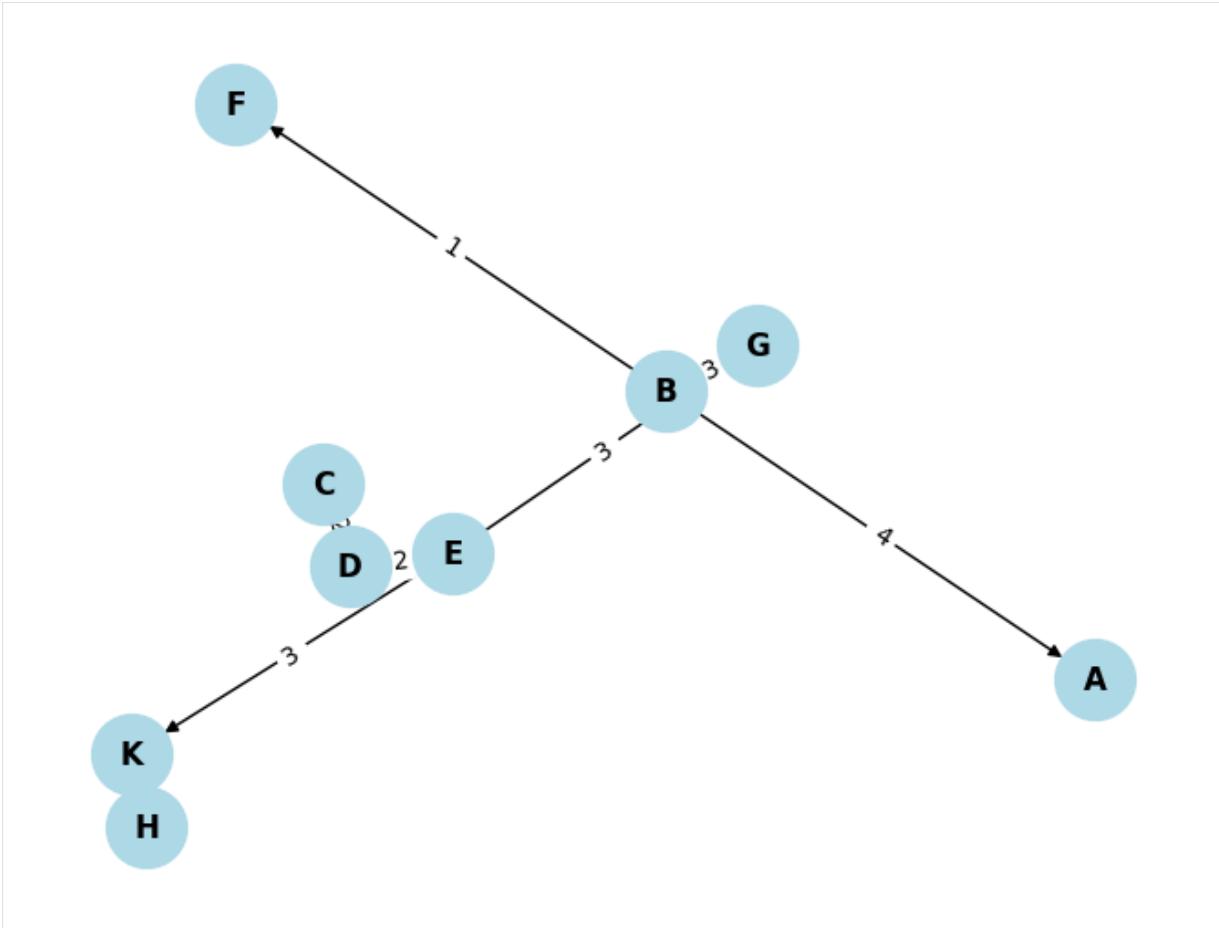
[7]: DIJKSTRA.draw()



[8]: prim(DIJKSTRA, "C").draw()



```
[9]: # the automated layout of graphs is semi-random
# you may need to re-run this cell several times to get a good layout
dijkstra(DIJKSTRA, "C").draw()
```



18.4.3 Applications

The following exercises show applications of the SSSP problem and Dijkstra's algorithm.

Exercise 18.4.2

Sometimes we want to solve the **single-pair shortest path** problem: find the shortest path between a given start and a given end node. How would you change Dijkstra's algorithm so that instead of a tree it returns a sequence of nodes representing the shortest path from the start to the end node? The end node is an extra input of the new algorithm. If the end node is not reachable from the start node, the output sequence is empty.

You don't have to rewrite the algorithm or the code: a brief description of the changes suffices.

Hint Answer

Exercise 18.4.3

Sometimes a problem setter has a vague idea of what they want and the problem solver has to provide a precise and implementable definition. This exercise presents such a scenario. It's a modelling exercise without one right answer.

You were *previously asked* by an advertising agency to determine the best train stations for displaying an advert, given an undirected unweighted graph of the rail network. Suppose the

edges are now weighted with the travel times.

The agency thinks the adverts will be seen by more people in central stations. You're asked to come up with a precise definition of which node or nodes in a weighted undirected connected graph are central and to outline an algorithm to find them. The algorithm doesn't have to be efficient.

Hint Answer

18.5 Summary

A **greedy algorithm** solves an optimisation problem incrementally, making at each step a greedy choice for the best available option. If the options are known in advance, they can be sorted from best to worst.

Greedy algorithms are usually very efficient but often are not correct: choosing one best option at a time doesn't necessarily lead to the best solution.

Finding a counter-example proves that an algorithm is incorrect. Counter-examples are often small and involve contrasting items. For greedy algorithms, counter-examples may also include several equally best options to choose from, with at least one of them not leading to an optimal solution.

The **interval scheduling problem** asks for a largest subset of non-overlapping intervals, given the start and end point of each interval. It can be solved in log-linear time by a greedy algorithm that sorts intervals by ascending end time and picks one at a time, if it doesn't overlap previously chosen intervals. Other greedy choices don't work, as shown by counter-examples.

18.5.1 Weighted graphs

A **weighted graph** has numbers, called weights, associated with its edges. Weights typically represent the cost, time or distance to travel between nodes. A shortest path between two nodes has the lowest sum of weights of all paths between those nodes.

Edge list, adjacency matrix and adjacency list representations can be adapted to weighted graphs. Classes `WeightedDiGraph` and `WeightedUndirectedGraph` use adjacency lists.

A **spanning tree** of an undirected graph is a subgraph that has all the graph's nodes but is a tree. A disconnected graph has no spanning tree. A connected graph may have several spanning trees.

A **minimum spanning tree (MST)** of a weighted undirected connected graph is a spanning tree with the lowest sum of weights. A graph may have multiple MSTs.

Prim's algorithm is a greedy algorithm that constructs an MST incrementally from a given start node. The greedy choice is to pick the lowest-weight edge connecting a node in the tree to a node not in the tree, which is thereby put in the tree.

The **single-pair shortest path problem** asks for a shortest path from a given start node to a given end node, if there is one. The **single-source shortest paths problem** asks to solve the single-pair problem for a given start node and each other node reachable from it.

Dijkstra's algorithm can solve the single-source problem for weighted graphs without negative weights. It's a modification of Prim's algorithm. The greedy choice is to pick the edge leading to

the next nearest node from the start node.

Prim's and Dijkstra's algorithms have worst-case complexity $O(e \log e)$.

18.5.2 Python

Function `abs` returns the absolute value of its numeric argument: `abs(x)` is $-x$ if x is negative, otherwise it's x .

CHAPTER 19

PRACTICE 2

This chapter introduces no new concepts, only new problems to practise algorithmic techniques and data structures, without you being told in advance which one to use. You should skim the advice in the *next chapter* before attempting the problems. They are in increasing difficulty order, in my opinion. Each problem can be solved with a technique or data structure introduced since Chapter 11.

The advice of *Chapter 9* applies again:

- Read all problems first, to decide in which order to tackle them.
- Don't worry if you don't finish them all: next week's TMA submission has priority.
- Compare approaches with your study buddy.
- Post alternative solutions to mine in the forums, with appropriate spoiler alerts.

This chapter's problems support the following learning outcomes.

- Develop and apply algorithms and data structures to solve computational problems – you will apply several of the ADTs and techniques you learned.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – you will be asked to describe your approach, before implementing it.
- Analyse the complexity of algorithms to support software design choices – you will have to suggest or evaluate alternative approaches.
- Write readable, tested, documented and efficient Python code – you will have to add tests and use Python data types and operations.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

19.1 Jousting

Once upon a time, in a far, far away land, jolly knights josted every day to practise killing marauding dragons that put damsels in distress. As the number of dragons alive decreased and the number of damsels who could defend themselves increased, the knights got idle and paid more attention to the king's politics. To avoid closer scrutiny of his actions, the king created a distraction: a prestigious tournament with a pompous title and a pointless trophy. That kept the knights jolly and busy.

To give his weak (but very rich and very supportive) cousin a fighting chance, the monarch changed the rules under the pretence of fairness, inclusivity and innovation. He decreed that jousts would be carried out one after the other, each between the two strongest knights still in the competition.

The monarch and his cousin reasoned that if the strong knights fight and eliminate each other, and get weaker in the process, the cousin, being among the last knights to fight, might become the champion. The calculating cousin prefers to not leave anything to chance and asks for a simulation of the tournament.

The input is a sequence of the knights' strengths, including the cousin's, in no particular order. The strengths are represented by positive integers.

The tournament is a succession of jousts, each between the two strongest remaining knights. When two knights of equal strength fight, they knock each other out: both are eliminated. Otherwise, the weaker knight is eliminated and the stronger one remains in the competition, but his strength is diminished by the opponent's strength. The output is the index of the knight who wins the tournament. If the final joust ends in a simultaneous knock-out (KO) of two equal-strength knights, the output is -1 .

For example, if the knights have strengths 5, 3, 6, 2, then the first and third knights joust first. The third knight wins, but his strength drops to $6 - 5 = 1$. The second and fourth knights fight next because now they're the strongest. The second knight wins and his remaining strength is $3 - 2 = 1$. Finally, the second and third knights joust but neither wins. The output is -1 .

19.1.1 Exercises

Exercise 19.1.1

Outline an algorithm to simulate the tournament. Indicate any necessary ADTs.

Hint Answer

Exercise 19.1.2

What's the worst-case scenario for the simulation? What is its complexity? State your assumptions, e.g. which data structures are used for the ADTs.

Hint Answer

Exercise 19.1.3

Add tests below and implement your algorithm.

```
[1]: from algoesup import test

def winner(strength: list) -> int:
    """Return the index of the winning knight or -1 if no one wins.

    Preconditions: all items in strength are positive integers
    """
    pass

winner_tests = [
    # case,           strength,      winner
    ('no one wins', [5, 3, 6, 2], -1),
    ('second wins', [5, 3, 6, 1], 1), # winner has initial_
→strength 3
    # your tests
]

test(winner, winner_tests)
```

Hint Answer

Once the cousin ran the simulation, he realised he wouldn't win and gave the king a lame excuse to chicken out of the tournament. According to legend, he later had an unfortunate encounter with a dragon. (Dragons do like chickens, especially roasted.)

19.2 Dot product

The dot product of two equal-length sequences of integers $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_n)$ is

$$X \cdot Y = x_1 \times y_1 + x_2 \times y_2 + \cdots + x_n \times y_n.$$

Given two sequences with $n > 0$, what's the smallest dot product we could obtain, if we could rearrange the order of the numbers within each sequence?

For example, the sequences $(1, 2, -1)$ and $(2, 6, 3)$ have dot product

$$(1, 2, -1) \cdot (2, 6, 3) = 2 + 12 + -3 = 11.$$

But if we rearrange one or both sequences, the lowest dot product we get is 1. Various rearrangements lead to it, including

$$(1, 2, -1) \cdot (3, 2, 6) = 3 + 4 + -6 = 1$$

and

$$(-1, 2, 1) \cdot (6, 2, 3) = -6 + 4 + 3 = 1.$$

As a further example, if the sequences are $(1, -1, 5, 0, 7)$ and $(3, 7, -9, 2, 0)$ then the lowest dot product is -68 , which can be obtained as follows, among other ways:

$$(1, -1, 5, 0, 7) \cdot (2, 7, 0, 3, -9) = 2 + -7 + 0 + 0 + -63 = -68.$$

19.2.1 Exercises

Exercise 19.2.1

1. Outline an exhaustive search algorithm to solve this problem.
2. What's the complexity for sequences of length n ?
3. Assuming the search generates and tests each candidate in a nanosecond, how long will it take for $n = 20$?

Conclude whether exhaustive search is a practical approach for this problem.

Hint Answer

Exercise 19.2.2

When I presented greedy choices for the *interval scheduling problem*, I provided an informal rationale for each one: why that choice seemed a good idea. The goal was to maximise the number of non-overlapping intervals, so for example I chose at each step the interval with the fewest overlaps to reduce the number of intervals eliminated at each step.

Present a greedy choice for the dot product minimisation problem and your rationale. You must explain which integer of X and which integer of Y you choose to multiply in each step, to attempt to minimise the overall sum.

Your greedy choice must produce the correct result for the above examples but you're not asked to prove it is correct for all inputs.

Answer

Exercise 19.2.3

Implement and test a greedy algorithm based on your greedy choice. Complete the function header and docstring.

```
[1]: from algoesup import test

def min_dot_product(x, y):
    pass

min_dot_product_tests = [
    # case,           x,                  y,          smallest
    ('n = 3',        [1, 2, -1],        [2, 6, 3],      1),
    ('n = 4',        [1, 2, -1, 4],     [2, 6, 3, 5],  1),
    ('n = 5',        [1, 2, -1, 4, 3],  [2, 6, 3, 5, 7], 1),
    ('n = 6',        [1, 2, -1, 4, 3, 5], [2, 6, 3, 5, 7, 9], 1),
    ('n = 7',        [1, 2, -1, 4, 3, 5, 2], [2, 6, 3, 5, 7, 9, 8], 1),
    ('n = 8',        [1, 2, -1, 4, 3, 5, 2, 6], [2, 6, 3, 5, 7, 9, 8, 10], 1),
    ('n = 9',        [1, 2, -1, 4, 3, 5, 2, 6, 7], [2, 6, 3, 5, 7, 9, 8, 10, 12], 1),
    ('n = 10',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14], 1),
    ('n = 11',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16], 1),
    ('n = 12',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18], 1),
    ('n = 13',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20], 1),
    ('n = 14',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22], 1),
    ('n = 15',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24], 1),
    ('n = 16',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26], 1),
    ('n = 17',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28], 1),
    ('n = 18',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], 1),
    ('n = 19',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32], 1),
    ('n = 20',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34], 1),
    ('n = 21',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36], 1),
    ('n = 22',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38], 1),
    ('n = 23',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40], 1),
    ('n = 24',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42], 1),
    ('n = 25',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44], 1),
    ('n = 26',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46], 1),
    ('n = 27',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48], 1),
    ('n = 28',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50], 1),
    ('n = 29',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52], 1),
    ('n = 30',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54], 1),
    ('n = 31',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56], 1),
    ('n = 32',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58], 1),
    ('n = 33',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60], 1),
    ('n = 34',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62], 1),
    ('n = 35',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64], 1),
    ('n = 36',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66], 1),
    ('n = 37',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68], 1),
    ('n = 38',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70], 1),
    ('n = 39',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72], 1),
    ('n = 40',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74], 1),
    ('n = 41',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76], 1),
    ('n = 42',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78], 1),
    ('n = 43',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80], 1),
    ('n = 44',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82], 1),
    ('n = 45',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84], 1),
    ('n = 46',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86], 1),
    ('n = 47',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88], 1),
    ('n = 48',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90], 1),
    ('n = 49',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92], 1),
    ('n = 50',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88], [2, 6, 3, 5, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94], 1),
    ('n = 51',       [1, 2, -1, 4, 3, 5, 2, 6, 7, 9, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60
```

(continued from previous page)

```
'n = 5', [1, -1, 5, 0, 7], [3, 7, -9, 2, 0], -68),
# your tests
]

test(min_dot_product, min_dot_product_tests)
```

Hint Answer

19.3 Beams

Bridges, buildings and other structures often use beams that are joined to form triangles, as they improve the stability of the structure. Given some points in the structure, and which ones are joined by beams, find the weak points, i.e. those that are not part of a triangle.

For example, the set of pairs $\{(1,2), (1,3)\}$ represents two beams, one joining points 1 and 2, the other joining points 1 and 3. This doesn't form a triangle, so all points are weak. Adding a beam to join points 2 and 3 would make all points strong.

Structural points that aren't mentioned in the input don't exist. For example, for input $\{(2,3), (3,7)\}$ report only that points 2, 3, and 7 are weak, and ignore points 1, 4, 5 and 6.

Function: weak points

Inputs: *beams*, a set of pairs of integers

Preconditions: for every pair (a, b) in *beams*, $a \neq b$

Output: *weak*, a set of integers

Postconditions: *weak* has exactly all integers occurring in *beams* that aren't part of a triangle

19.3.1 Exercises

Exercise 19.3.1

Both examples above each comprise two beams with a common point. They don't form a triangle and all three points are weak. Which further problem instances should be tested? You don't have to write a test table: just describe the problem instances.

Hint Answer

Exercise 19.3.2

Which ADT(s) would you use to represent a structure made of points and beams?

Hint Answer

Exercise 19.3.3

Outline an algorithm to solve the problem. It doesn't have to be efficient. You may wish to draw some points and beams to help think how an algorithm can detect the weak points.

Hint Answer

Exercise 19.3.4

What are the best- and worst-case scenarios for your algorithm? What are their complexities?

Hint Answer

Exercise 19.3.5

Add tests according to your answers to Exercises 19.3.1 and 19.3.4, i.e. make sure the tests also include best- and worst-case inputs. Write and run the function.

```
[1]: from algoesup import test

def weak_points(beams: set) -> set:
    """Return the points that aren't part of any triangle.

    beams is a set of pairs of integers.
    The output is a set of integers.
    Each integer represents a point.

    Preconditions: for every pair (a, b), a ≠ b
    Postconditions: the output only has points occurring in beams
    """
    pass

weak_points_tests = [
    # case,           beams,           weak points
    ('no triangle',   {(1, 2), (3, 1)}, {1, 2, 3}),
    ('missing points', {(7, 3), (3, 2)}, {2, 3, 7}),
    # your tests:
]

test(weak_points, weak_points_tests)
```

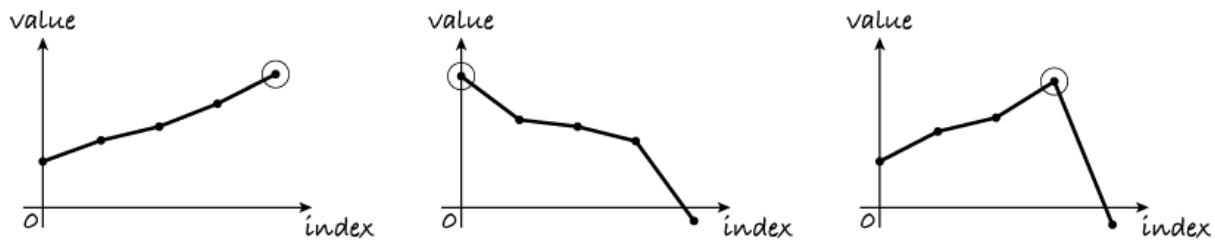
Answer

19.4 Up and down

Find the maximum of a given non-empty sequence of integers, knowing that they first ascend and then descend and that there are no consecutive repeated integers. The ascending or the

descending part may be empty: see the tests below. The next diagram shows the three possible shapes of the sequence, if we plot each integer at each index: only up, only down, up and down. The circles highlight where the maximum is in each case.

Figure 19.4.1



Sequences like $(-3, -1, -1, 0)$ aren't problem instances because they have consecutive repeated numbers, but sequences like $(-3, -1, 0, -1)$ are allowed.

19.4.1 Exercises

Exercise 19.4.1

Which algorithmic technique can be applied to efficiently solve this problem?

Hint Answer

Exercise 19.4.2

Outline an algorithm.

Hint Answer

Exercise 19.4.3

Complete the cell below and run it. Don't forget to complete the function header and to add a docstring.

```
[1]: from algoesup import test

def max_up_down(numbers):
    pass

max_up_down_tests = [
    # case,           numbers,          largest
    ('1 number',     [-1],             -1),
    ('only descending', [3, 2],         3),
    ('only ascending', [-3, -1, 0],    0),
    ('up and down',   [-3, -1, 0, 3, 2, -1], 3),
]
test(max_up_down, max_up_down_tests)
```

Hint Answer

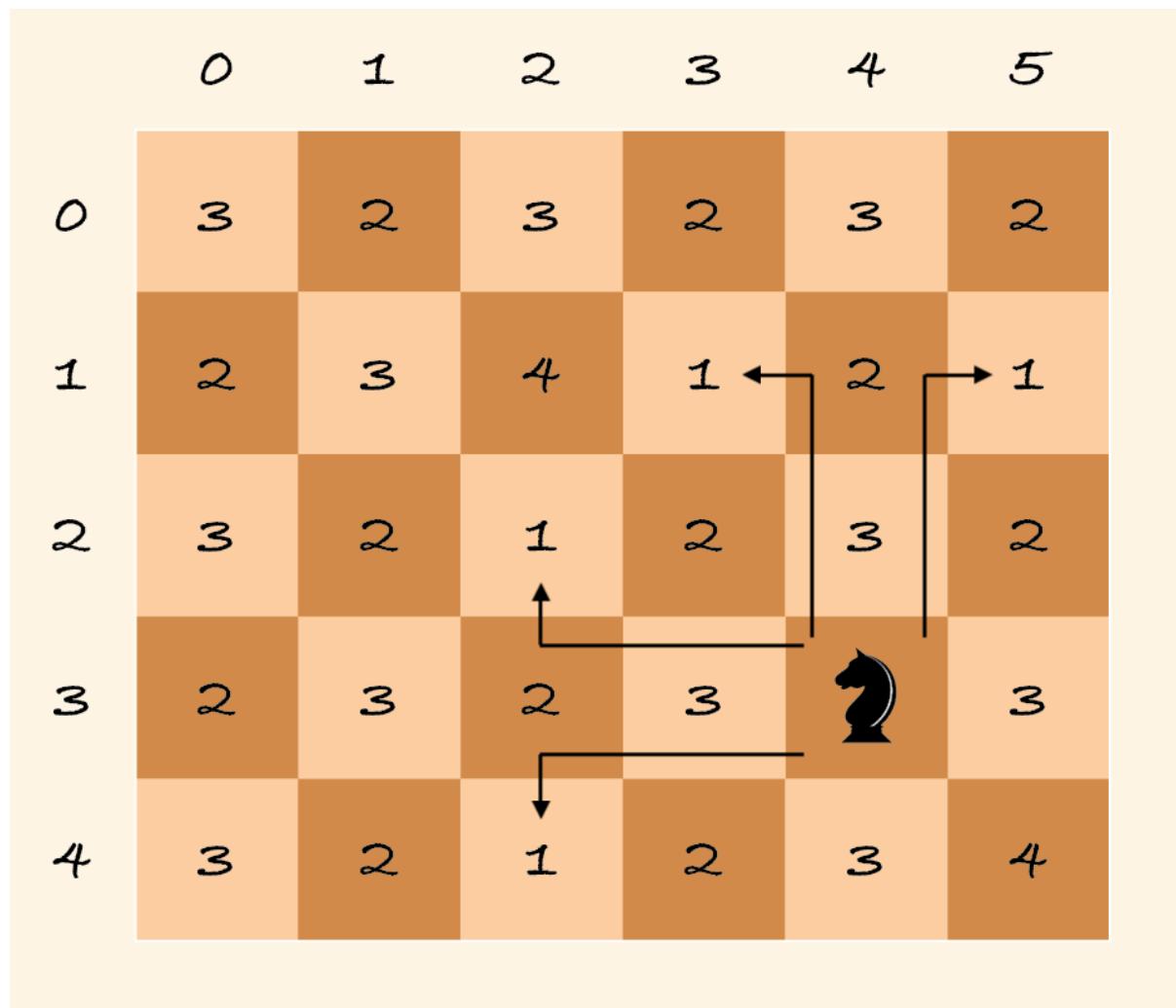
19.5 A knight goes places

And to finish off, another problem about jolly knights.

In the game of chess, the board is a grid of squares and the knight is a piece that moves in an L-shaped way: it moves two squares vertically and one square horizontally, or two squares horizontally and one square vertically.

The next figure shows a knight in row 3, column 4, and how many moves it takes to reach each square. The L-shaped lines with arrows show the four possible first moves.

Figure 19.5.1



For example, the knight can reach the bottom left-hand corner in three moves:

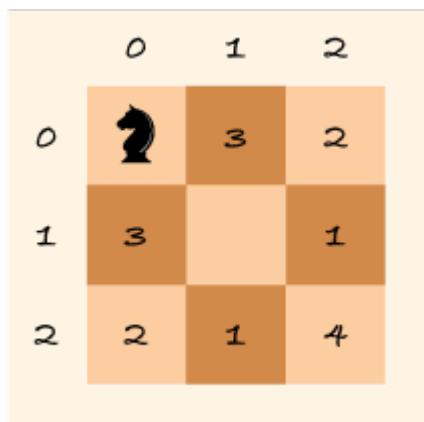
1. move two squares up and one to the left to row 1, column 3
2. move two squares left and one square down to row 2, column 1
3. move two squares down and one square left to row 4, column 0.

The second move could have been to row 3, column 2 instead.

Given the number of rows and columns, and the start and end squares of the knight, we want to know the smallest number of moves needed to reach the end square. For example, if the start and end squares are the same, the output is zero. You can assume the board has at least one square and that the start and end squares exist.

The knight cannot move outside the board and then back inside, so for some inputs the end square may not be reachable. In such cases the output should be -1 . For example, a knight cannot reach the middle square of a 3×3 board, unless it starts there.

Figure 19.5.2



19.5.1 Exercises

Exercise 19.5.1

Describe some edge cases that should be tested.

Hint Answer

Exercise 19.5.2

What ADT(s) will you use to represent the board and the knight's movements?

Hint Answer

Exercise 19.5.3

Outline an algorithm to solve the problem.

Hint Answer

Exercise 19.5.4

Implement and test your algorithm.

```
[1]: from algoesup import test
```

```
def knight_moves(size: tuple, start: tuple, end: tuple) -> int:
```

(continues on next page)

(continued from previous page)

```
"""Return least number of knight moves from start to end.
```

```
Return -1 if end is not reachable from start.
```

Preconditions:

- size is a pair (rows, columns) with rows > 0 and columns > 0
- start and end are pairs (r, c) with 0 <= r < rows and 0 <= c < columns

```
"""
```

```
pass
```

```
knight_moves_tests = [  
    # case,           size,      start,      end,      moves  
    ('1x1 board',     (1, 1),    (0, 0),    (0, 0),    0),  
    ('1 row, 2 cols', (1, 2),    (0, 0),    (0, 1),    -1),  
    ('2 rows, 1 col', (2, 1),    (1, 0),    (0, 0),    -1),  
    ('start = end',   (3, 3),    (1, 1),    (1, 1),    0),  
    ('bottom left',   (5, 6),    (3, 4),    (4, 0),    3), # figure 19.  
    ↪5.1  
    ('bottom right',  (5, 6),    (3, 4),    (4, 5),    4), # figure 19.  
    ↪5.1  
    ('3x3 to centre', (3, 3),    (0, 0),    (1, 1),    -1), # figure 19.  
    ↪5.2  
]  
  
test(knight_moves, knight_moves_tests)
```

Hint Answer

CHAPTER 20

TMA 02 PART 2

This study-free week is for you to catch up if you need to, and to complete and submit TMA 02.

This chapter provides some advice that complements the summaries of Chapters [16 \(trees\)](#), [17 \(graphs\)](#) and [18 \(greed\)](#). The guidance in Chapters [5](#), [10](#) (in particular about checking your TMA) and [15](#) still applies.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

20.1 Using graphs

If the word ‘network’ appears in the problem statement or there’s a relationship (e.g. dependency) between pairs of items, consider using a graph. The relationship may be explicit in the problem statement (e.g. friendship) or it may be implicit (e.g. being adjacent in a 2D grid).

20.1.1 Modelling with graphs

If the relationship is symmetric (whenever A is related to B, B is related to A), use an undirected graph; otherwise, use a directed graph.

If the relationship can be quantified (e.g. distance to the adjacent node) or has an associated numeric property (e.g. cost, duration), use a weighted graph. If the weight represents the distance between two points ‘as the taxi drives’ on a grid, then it’s the [Manhattan distance](#). If the weight represents the distance between two points ‘as the crow flies’ on a straight line, then it’s the Euclidean distance.

To choose a data structure to represent the graph consider whether it’s dense (use an adjacency matrix) or sparse (use an adjacency map).

When asked to outline an algorithm and its ADTs, if you’re using a graph, don’t forget to state:

- what the nodes and edges represent, e.g. airports and direct flights
- if the edges are directed and what determines a node being the source or target of the edge, e.g. if the formula in A depends on the value in B, then the edge goes from B to A

- if the edges are weighted and what the weights represent, e.g. travel cost
- if relevant to the problem, whether the graph is dense or sparse, cyclic or acyclic, connected or disconnected.

20.1.2 Algorithms

Algorithms on graphs must not rely on any particular order of the nodes and their neighbours.

If the problem is about reaching nodes, consider a depth- or breadth-first traversal of the graph. If the problem asks to minimise the number of ‘hops’ to go from one node to another, use BFS.

If the problem asks to minimise the total weight (e.g. cheapest or fastest route), then consider using Dijkstra’s shortest path algorithm or Prim’s minimum spanning tree algorithm. Remember that Dijkstra’s algorithm doesn’t work with negative weights.

Some problems can’t be solved by a direct application of an existing algorithm: they may require some creative adaptation of the algorithm.

Aim to state the complexity of a graph algorithm in terms of n and e , the number of its nodes and edges. For a dense graph, you may replace e by n^2 . An empty or complete graph is often a best or worst case for graph algorithms. Remember that if an algorithm goes through all neighbours of each node, it’s going through all edges.

20.2 Applying greed

If you have an optimisation problem asking for a single best solution, and the solution is a collection (e.g. a set of items), consider a greedy algorithm.

It starts with an empty collection and extends it by one item at a time, making a greedy choice of which item to add next.

20.2.1 Approaches

If you know in advance the possible items to put in the solution, like in the interval scheduling and knapsack problems, consider sorting them from best to worst choice, and adding them in that order.

1. let *solution* be an empty collection
2. let *extensions* be the items sorted from best to worst
3. for each *item* in *extensions*:
 1. if *item* is compatible with *solution*:
 1. add *item* to *solution*

If the items aren’t known in advance, the extensions have to be computed as the solution is created.

1. let *solution* be an empty collection
2. repeat:

1. let *extensions* be the possible next items compatible with *solution*
2. if *extensions* isn't empty:
 1. add one of the best of *extensions* to *solution*
 3. until *extensions* is empty

The following combination of both approaches keeps the extensions sorted in a priority queue, so that finding the best extension in step 2.2.1 is trivial, and updates the extensions according to which one was added to the solution, instead of recomputing all extensions from scratch in step 2.1.

1. let *solution* be an empty collection
2. let *extensions* be a queue with the items compatible with *solution*, with priority from best to worst
3. while *extensions* isn't empty:
 1. remove the first *item* from *extensions*
 2. add *item* to *solutions*
 3. update *extensions* according to *item*

Dijkstra's algorithm for the single-source shortest paths problem and Prim's algorithm for the minimum spanning tree problem use the last approach. The *solution* is a tree (a collection of nodes and edges), extended by one *item* (an edge to a new node) at a time. Step 3.3 looks at the neighbours of the new node to update the *extensions* (a priority queue of edges).

20.2.2 Correctness

Greedy algorithms are correct only if the immediate choice for the best option leads to the overall best solution. That's often not the case.

To prove that a particular greedy choice won't produce the correct output, you only need provide *one* counter-example, i.e. an input for which the greedy choice will not lead to the best solution for that input.

The counter-example should be small so that you can compute by hand both the output produced by the greedy choice and the correct output, to show they differ.

If at any time there are multiple best extensions, the greedy algorithm may choose the wrong one, which won't lead to the best solution. You may thus wish to think of a counter-example that leads to such a choice and assume the algorithm will take the wrong one.

A TMA question may ask you to outline or implement a greedy algorithm, including devising your own greedy choice, but it won't ask you to prove the algorithm is correct. The 'correctness' of your algorithm will be based on passing the tests.

20.3 Python

Unless a TMA 02 question explicitly states otherwise, your code can only use the types, classes, methods, functions, statements, constants and code templates introduced in Chapters 2–19 of this book. Here's a non-comprehensive index of them, in addition to Section [10.4](#).

20.3.1 Language constructs

- nested functions ([11.3.3](#))
- types `Callable` ([14.3.2](#)) and `Hashable` ([17.6.1](#)) in module `typing`

20.3.2 Standard library

- function `super()` ([17.6.2](#))
- function `abs` ([18.2](#))
- method `pop` on sets ([11.2.2](#))
- functions `sqrt` ([11.2.3](#)) and `factorial` ([11.4.3](#)) in module `math`
- functions `permutations` ([11.4.4](#)) and `combinations` ([11.5.4](#)) in module `itertools`
- key argument for function `sorted` and method `sort` ([14.1.4](#))
- functions `shuffle` ([14.6.3](#)) and `random` ([17.6.4](#)) in module `random`
- functions `heappush` and `heappop` in module `heapq` ([16.6.5](#))

20.3.3 M269 library

Recursion (file `m269_rec_list.py`):

- functions `is_empty`, `head`, `tail` ([12.3.2](#)) and `prepend` ([12.5.1](#))

Binary tree (file `m269_tree.py`):

- class `Tree` ([16.1.2](#))
- functions `is_empty`, `join`, `leaf`, `is_leaf` ([16.1.2](#)), `size` ([16.2.1](#)) and `write` ([16.3.2](#))
- constants `THREE`, `FOUR`, `FIVE`, `SIX`, `TPM`, `PMT`, `MPT` (example trees, [16.1.2](#))

Sorting (file `m269_sorting.py`):

- functions `suit`, `value`, `suit_value`, `identity` ([14.1.2](#))

Directed graphs (file `m269_digraph.py`):

- class `DiGraph` ([17.6.1](#))
- functions `bfs` ([17.8.1](#)) and `dfs` ([17.8.2](#))
- class `WeightedDiGraph` ([18.2.2](#))
- function `dijkstra` ([18.4.2](#))

Undirected graphs:

File m269_ungraph.py:

- class UndirectedGraph ([17.6.2](#))
- class WeightedUndirectedGraph ([18.2.2](#))
- function prim ([18.3.3](#))

File m269_graphs.py:

- functions null_graph, path_graph, cycle_graph, complete_graph ([17.6.3](#))
- function random_graph ([17.6.4](#))
- constants EMPTY_UG ([17.6.3](#)), RHOMBUS ([18.3.3](#)) and DIJKSTRA ([18.4.2](#))

CHAPTER 21

GRAPHS 2

This chapter further illustrates the versatility of graphs and how general abstract algorithms on graphs can be used to solve concrete practical problems.

Section 17.1 on modelling with graphs mentioned that graphs can represent transport networks. These are usually connected: one can reach every node from any other node. However, that may not be the case after disruptions like road flooding and flight cancellations. In the extreme case, the network breaks down into separate, disconnected subnetworks and travel from one subnetwork to another becomes impossible. Sections 1 and 2 of this chapter are about the problem of finding the nodes that are in their own separate subgraph and therefore cut off from other nodes.

Section 17.1 also noted that directed graphs can represent dependency networks, like which spreadsheet cells depend on which other ones. Section 3 of this chapter introduces an algorithm that determines the order in which cells must be evaluated so that no cell is evaluated before any cell it depends on.

Section 17.1 included an example of a state transition graph for the Noughts and Crosses game. The nodes represent the states of the board; the edges represent the possible player moves. Section 4 of this chapter shows that problems that look complicated may be solved with a standard graph algorithm if we model those problems with state transition graphs.

This chapter supports these learning outcomes:

- Develop and apply algorithms and data structures to solve computational problems – you will learn that some problems can be solved with general-purpose graph algorithms by transforming the input into a graph.
- Analyse the complexity of algorithms to support software design choices – you will be asked to determine the complexity of alternative algorithms.

You should first read again the summaries of Chapters [17](#) and [18](#) to refresh your memory about graph-related concepts and algorithms needed for this chapter.

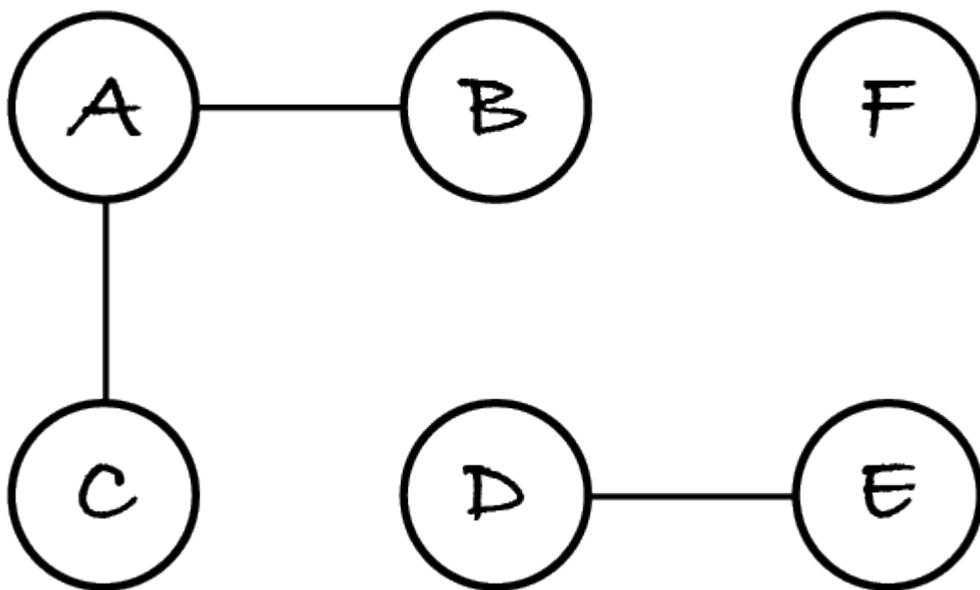
Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

21.1 Undirected graph components

Sometimes we want to know if there's a path between two given nodes, but the actual path, if it exists, is of little relevance. For example, if bad weather brings power cables down, we want to know for each residential or industrial area if it's connected to a generator. We don't need the path from the generator to the area: we only want to know whether it exists, to determine the cut-off areas.

The more general problem is to compute, for a given graph, its various separate subgraphs of mutually reachable nodes. The problem is slightly different for undirected and directed graphs, so let's first look at the former. Consider the following undirected graph.

Figure 21.1.1



It has three subgraphs of connected nodes. One subgraph consists of nodes A, B, C and their edges, another of nodes D and E and their edge, and the third has node F by itself. Each of these subgraphs is called a **connected component** of the whole graph.

A component of an undirected graph is a largest possible connected subgraph. For example, edge A–B and its nodes aren't a component: while they form a connected subgraph, it isn't as large as possible because we can add node C and edge A–C to get a larger connected subgraph. There are no further edges for nodes A, B and C, so they (and their edges) form a component: there's no larger connected subgraph they're part of.

In other words, a connected component of an undirected graph is a largest set of nodes that are mutually reachable together with all their edges. For example, if the above graph had edge B–C, the component would be subgraph A–B–C–A: omitting any of the edges between those nodes would lead to a subgraph that isn't the largest possible.

21.1.1 Problem definition and instances

The general problem we want to solve is: given an undirected graph and two nodes A and B, is there a path between them? Are they mutually reachable? In the power grid example, we want to know if electricity can flow from a generator A to a residential or industrial estate B and back.

Since components are subgraphs of mutually reachable nodes, to answer the question we compute the components and then check if A and B are in the same component. For example, in the graph above, A and E are in different components, so they're not mutually reachable.

We need an ADT that associates each node with the component that it's in: that's the map ADT. The easiest way to label the components is to number them from 1 onwards. Here's the precise problem definition.

Function: connected components

Inputs: *graph*, an undirected graph

Preconditions: true

Output: *component*, a map of objects to integers

Postconditions:

- *component* maps the nodes of *graph* to the integers from 1 to the number of connected components in *graph*
- *component(a) = component(b)* if and only if nodes *a* and *b* are mutually reachable

To create some graphs for testing, I must first load the necessary code. Since breadth- and depth-first search use queues and stacks, it's safest to always load the queue and stack implementations too, even if we end up not traversing graphs.

```
[1]: %run -i ../m269_digraph
%run -i ../m269_ungraph
%run -i ../m269_queue
%run -i ../m269_stack
```

Let's create the above graph to later test the computation of components.

```
[2]: undirected = UndirectedGraph()
for node in "ABCDEF":
    undirected.add_node(node)
undirected.add_edge("A", "B")
undirected.add_edge("A", "C")
undirected.add_edge("D", "E")
```

We also need to test with edge cases: a graph with the fewest possible components and a graph with the most components. Describe two such graphs and how many components they have.

A null graph (without edges) has the most components: one per node. A connected graph has the fewest components: only one.

We already have functions to *generate null and connected graphs*. We'll use them later to create problem instances for testing.

21.1.2 Algorithm and complexity

The key idea to compute the components is that traversing an undirected graph from node A visits all nodes reachable from A, and therefore visits all nodes in the same component as A.

To compute all components, we repeatedly traverse the graph from each node that hasn't been assigned to a component yet. Each traversal adds the nodes it visits to a new component. Here's an outline of the algorithm:

Create an empty map. Initialise a component counter with 1. Go through each node in the graph. If the node is in the map, we already know its component, so do nothing. Otherwise, do any graph traversal from that node. Add all nodes returned by the traversal to the map, associated to the current component counter. Then increment the counter. After going through all nodes, return the map.

The complexity can be analysed as follows. Remember that n and e refer to the number of nodes and edges in a graph.

- Checking if a node is in the map takes constant time if the map is implemented with a hash table. So checking all n nodes takes $\Theta(n)$.
- The counter increments take $\Theta(1)$ in the best case and $\Theta(n)$ in the worst case, because that's the least and most number of components, as seen earlier.
- Each traversal only visits part of the graph, but together the traversals visit every node and edge once. They're equivalent to a single traversal of the whole graph, which has complexity $\Theta(n + e)$ ([Section 17.7.2](#)).

Considering only the fastest-growing term, we can say that the complexity of the algorithm is $\Theta(n + e)$. This is both the best- and worst-case complexity, because the whole graph is traversed to find all components.

21.1.3 Code and tests

In translating the algorithm outline to Python we must remember that our *traversal functions* return the tree of all paths from the start node. We must add each tree node to the map.

Any traversal will work. The code below uses depth-first search (DFS). You can replace it with breadth-first search (BFS) and confirm you get the same components.

```
[3]: # this code is also in m269_ungraph.py
```

```
def connected_components(graph: UndirectedGraph) -> dict:
    """Return the connected components of graph.

    Postconditions: the output maps each node to its component,
    numbered from 1 onwards.
    """
    component = dict()
    counter = 1
    for node in graph.nodes():
```

(continues on next page)

(continued from previous page)

```

if node not in component:
    tree = dfs(graph, node)
    for reached in tree.nodes():
        component[reached] = counter
    counter = counter + 1
return component
    
```

Let's test the code with the example graph.

```

[4]: connected_components(undirected)

[4]: {'E': 1, 'D': 1, 'C': 2, 'B': 2, 'A': 2, 'F': 3}
    
```

As expected, it finds the three components: nodes A, B and C, nodes D and E, and node F by itself.

Let's test with the edge cases. Remember that the graphs that `m269_graphs.py` generates have nodes $0, 1, 2, \dots, n - 1$.

```

[5]: %run -i ../m269_graphs

# most components: no node has neighbours
connected_components(null_graph(5))

[5]: {0: 1, 1: 2, 2: 3, 3: 4, 4: 5}

[6]: # fewest components: a connected graph; could be cycle graph or
      →complete graph
connected_components(path_graph(5))

[6]: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
    
```

As expected, every node of the null graph is in a separate component and all nodes of a connected graph are in the same component.

Exercise 21.1.1

This exercise asks you to apply your knowledge of how connected components are computed to create a bespoke algorithm for a particular problem.

Consider again the power grid example. Implement the next function, which returns the set of nodes not connected to any power source node.

```

[7]: def disconnected(graph: UndirectedGraph, sources: set) -> set:
    """Return all nodes not connected to any of the sources.

    Preconditions: sources is a non-empty subset of the graph's nodes
    """
    pass
    
```

(continues on next page)

(continued from previous page)

```
disconnected(undirected, { "A" }) # you should obtain { 'D', 'E', 'F' }
```

Hint Answer

Exercise 21.1.2

The following is an exercise in modelling a situation. There's no right or wrong answer.

A government agency wants to know which train stations are critical: that is, which stations would cause the most disruption if, due to accident or incident, they had to be closed and no trains could start, terminate or pass through those stations.

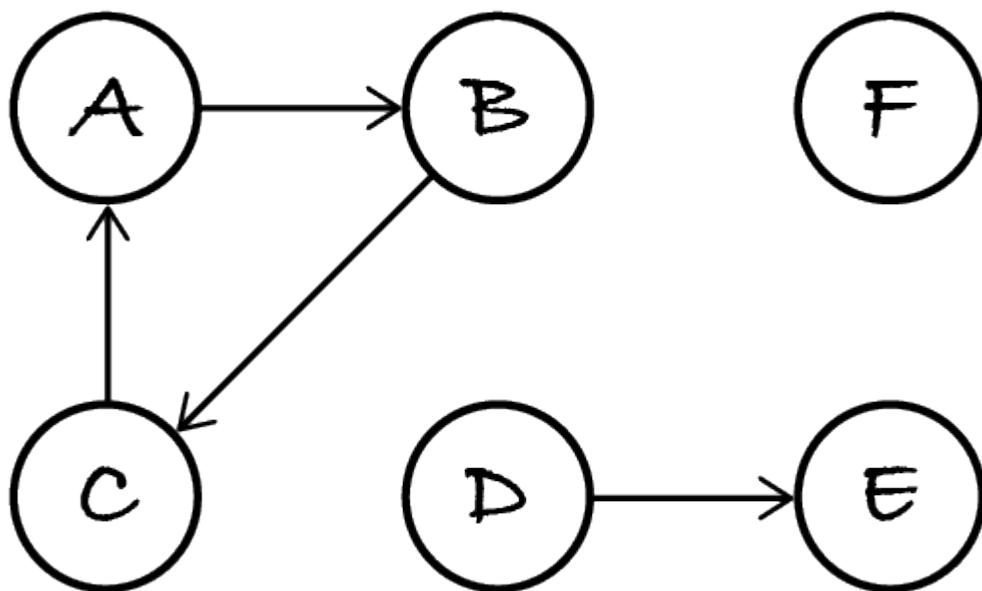
Given a connected undirected graph representing the train network, how would you define the critical nodes, based on the notion of components?

Hint Answer

21.2 Directed graph components

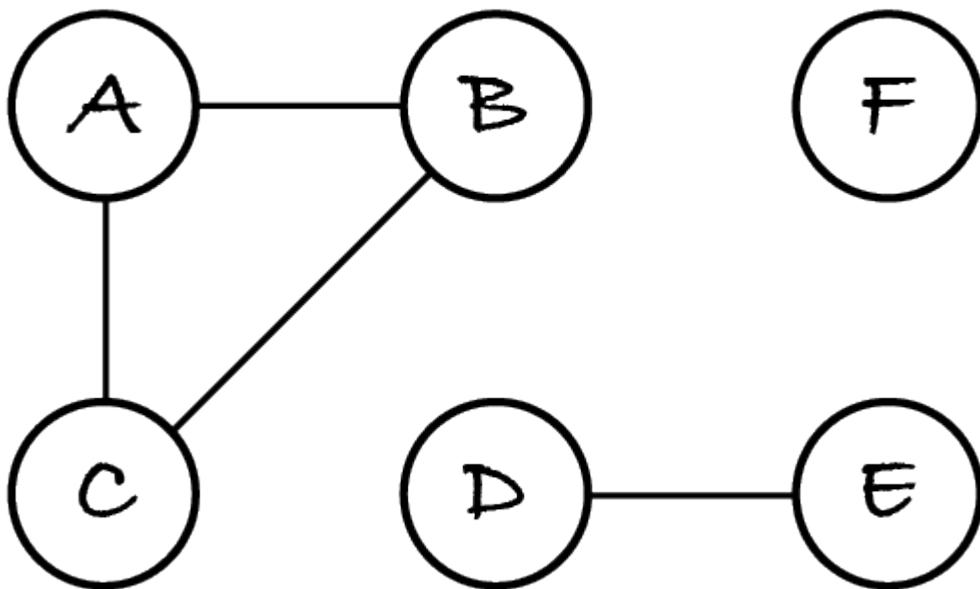
Contrary to undirected graphs, digraphs have two kinds of components. Consider the following example.

Figure 21.2.1



If we ignore the edge directions then we get this undirected graph, with three connected components:

Figure 21.2.2



The **weakly connected components** of a digraph G are the connected components of the undirected version of G. So, the above digraph has three weakly connected components: nodes A, B, C and their three directed edges; nodes D and E and their edge; node F. Nodes of weakly connected components are mutually reachable if we ignore the edge directions.

By contrast, a **strongly connected component** is a largest subgraph where all nodes are mutually reachable when considering the edge directions. Nodes A, B and C are mutually reachable because their edges form a cycle. Each other node forms a strongly connected component by itself, because the other nodes can only reach themselves, via paths of length zero. To sum up, the above digraph has three weakly and four strongly connected components.

Just to check your understanding, how many weakly and strongly connected components does digraph $A \rightarrow B \leftarrow C$ have?

It has one weakly connected component (the whole graph) and three strongly connected components (each node by itself).

21.2.1 Problem and instances

Before we turn to the problem of computing components in digraphs, here's the digraph in Figure 21.2.1, for testing.

```
[1]: %run -i ./m269_digraph
%run -i ./m269_queue
%run -i ./m269_stack

digraph = DiGraph()
for node in "ABCDEF":
    digraph.add_node(node)
for edge in ("AB", "BC", "CA", "DE"):
    digraph.add_edge(edge[0], edge[1])
```

I will compute the strongly connected components and leave the weakly connected components to you.

Exercise 21.2.1

Complete the following function.

```
[2]: def weakly_connected_components(graph: DiGraph) -> dict:  
    """Return the weakly connected components of graph.  
  
    Postconditions: the output maps each node to its component,  
    numbered from 1 onwards.  
    """  
  
    pass
```

weakly_connected_components(digraph)

You should obtain three components: A, B and C; D and E; F.

Hint Answer

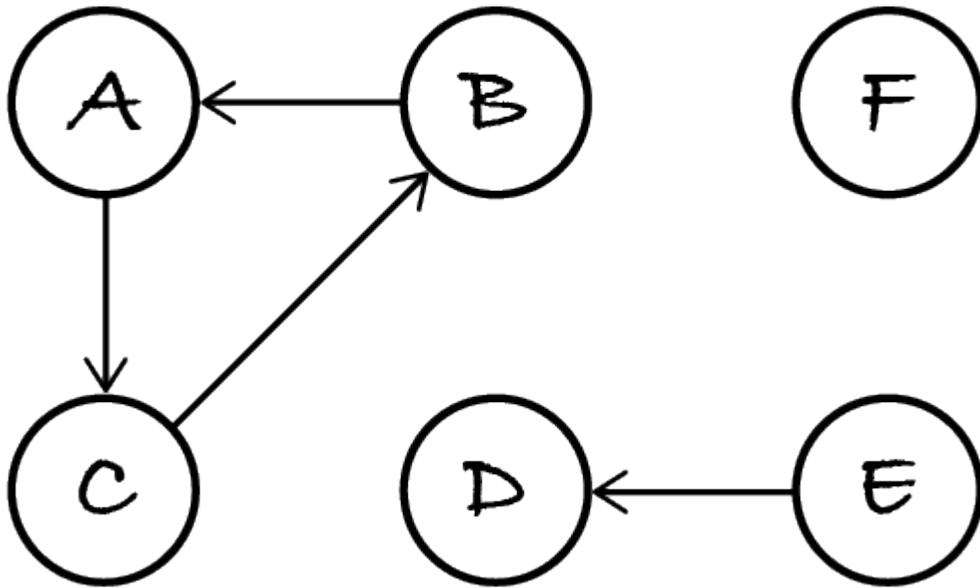
21.2.2 Algorithm and complexity

The algorithm for the strongly connected components is similar to the algorithm for connected components in undirected graphs: for each node A that isn't yet in the map, we find the nodes that are in the same component as A and add them to the map.

Node B is in the same component as A if there's a path from A to B and a path from B to A. If we compute the set of all nodes that have paths *from* A and the set of all nodes that have paths *to* A, then the intersection of both sets is the set of nodes in the component of A, because those nodes have a path from and to A.

Traversal algorithms follow the directions of the edges and hence compute the paths *from* a given node A. One way to compute all paths *to* A is to reverse the direction of all edges to obtain the **reverse graph**, and then compute the paths *from* A in the reverse graph. As mentioned in [Section 17.1](#), if a digraph represents the ‘follows’ relationship on Twitter, then its reverse graph represents the ‘is followed by’ relation. Here is the reverse graph of our example.

Figure 21.2.3



Info: The reverse graph is also called the transpose graph because it can be obtained by transposing the adjacency matrix.

If we traverse the original graph from A, we obtain the tree A → B → C and thereby the nodes that can be reached from A. If we traverse the reverse graph from A, we obtain the tree A → C → B, and thereby the nodes that can reach A in the original graph. Nodes A, B and C are in both trees, so they all can reach A and be reached from A, which means they form a strongly connected component.

As a further example, if we traverse the original graph from D, we obtain tree D → E. If we traverse the reverse graph from D, we obtain tree D (a single node). Both trees have only D in common, so it forms a strongly connected component by itself.

To sum up: for each node V that isn't yet in the map, we find which nodes are reachable from V (using the input graph) and from which nodes we can reach V (using the reverse graph). The nodes in both sets are by definition in the same strongly connected component as V. Here's the algorithm, using again a depth-first traversal, but any kind of traversal will do.

1. let *reverse graph* be the reverse of *graph*
2. let *component* be an empty map
3. let *current* be 1
4. for each *node* in *graph*:
 1. if *node* not in *component*:
 1. let *forward* be the nodes of DFS(*node*, *graph*)
 2. let *backward* be the nodes of DFS(*node*, *reverse graph*)
 3. for each *common* in *forward* intersected with *backward*:

1. let *component*(*common*) be *current*
4. let *current* be *current* + 1

Step 1 always has complexity $\Theta(n + e)$. In the worst case:

- steps 4.1.1 to 4.1.4 are executed *n* times
- steps 4.1.1 and 4.1.2 visit together the whole graph in $\Theta(n + e)$
- step 4.1.3 takes $\Theta(n)$ to compute the intersection of two node sets.

The total complexity is

$$\Theta(n + e) + n \times (\Theta(n + e) + \Theta(n)) = n \times \Theta(n + e) = \Theta(n^2 + ne)$$

by considering only the fastest growing part of the expression.

In the best case, the loop executes only once, adding all nodes to the map. (This means the best-case scenario is when the graph has only one strongly connected component.) Replacing the worst-case *n* iterations with a best-case single iteration in the previous formula, we get the best-case complexity:

$$\Theta(n + e) + 1 \times (\Theta(n + e) + \Theta(n)) = \Theta(n + e).$$

21.2.3 Code and tests

Here's the code for reversing a digraph and computing its strongly connected components.

```
[3]: # this code is also in m269_digraph.py

def reverse(graph: DiGraph) -> DiGraph:
    """Return the same graph but with edge directions reversed."""
    result = DiGraph()
    for node in graph.nodes():
        result.add_node(node)
    for edge in graph.edges():
        result.add_edge(edge[1], edge[0])
    return result

def strongly_connected_components(graph: DiGraph) -> dict:
    """Return the strongly connected components of graph.

    Postconditions: the output maps each node to its component,
    numbered from 1 onwards.
    """
    reverse_graph = reverse(graph)
    component = dict()
    counter = 1
```

(continues on next page)

(continued from previous page)

```

for node in graph.nodes():
    if node not in component:
        forward = dfs(graph, node).nodes()
        backward = dfs(reverse_graph, node).nodes()
        for common in forward.intersection(backward):
            component[common] = counter
        counter = counter + 1
return component
    
```

Let's test the code with the example digraph.

```

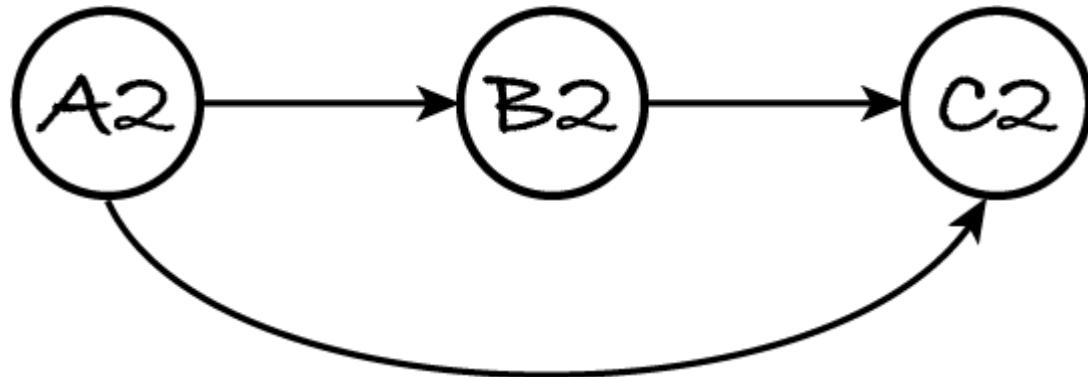
[4]: strongly_connected_components(digraph)
[4]: {'E': 1, 'D': 2, 'C': 3, 'B': 3, 'A': 3, 'F': 4}
    
```

As expected, nodes A, B, C are in one component and each other node is in its own component.

21.3 Topological sort

I mentioned in [Section 17.1](#) that digraphs can represent scheduling constraints: an edge $A \rightarrow B$ states that task or event A must occur before task or event B. The example was the evaluation of formulas in spreadsheet cells. Here again is the cell dependency digraph. If cell A2 changes, the spreadsheet must first re-evaluate B2 and only then C2 because C2 depends on A2 and B2.

Figure 21.3.1



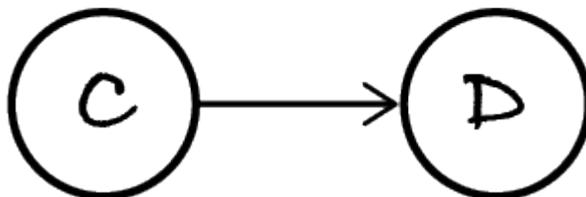
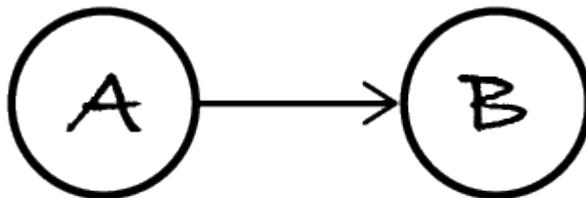
21.3.1 Problem

Given a digraph, we want to know a possible schedule that indicates in which order to do the tasks or carry out the events represented by the nodes. Such a schedule is called a **topological sort** of the digraph: it's a sequence of the graph's nodes so that for every edge $A \rightarrow B$, node A appears before node B in the sequence. If you can lay out a digraph so that every edge points from left to right, then a topological sort is obtained by reading the nodes from left to right.

The spreadsheet graph above is laid out from left to right and so has topological sort (A2, B2, C2). That's the only possible topological sort. For example, sequence (A2, C2, B2) isn't a topological sort because C2 comes before B2, contrary to the order imposed by edge B2 \rightarrow C2.

Some digraphs have multiple topological sorts. For example,

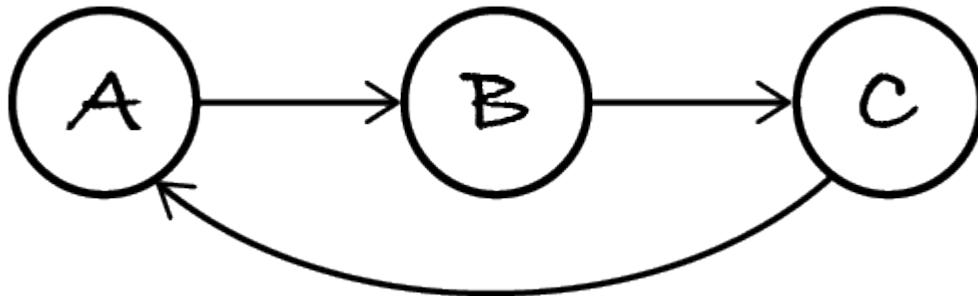
Figure 21.3.2



has topological sorts (A, B, C, D), (A, C, B, D), (C, A, B, D), etc. Only the node permutations where B appears before A or D appears before C, like (B, A, C, D) and (D, A, B, C), aren't topological sorts.

A cyclic digraph, like the following one, has no topological sort.

Figure 21.3.3



Any ordering of the nodes will go against one of the edges. For example, (A, B, C) isn't a topological sort because C must appear before A due to edge C → A. No matter which node we choose to start the sequence, some other node must appear before it, so no topological sort is possible. There's also a visual explanation. A graph with a cycle can't be laid out with all edges pointing left to right, because there's always a right-to-left edge to close the cycle.

In summary, every acyclic digraph has at least one topological sort. We want to compute one of them, it doesn't matter which.

Function: topological sort

Inputs: *graph*, a digraph

Preconditions: *graph* is acyclic

Output: *schedule*, a sequence of objects

Postconditions:

- *schedule* is a permutation of the *graph*'s nodes
- for every edge A → B in *graph*, A appears before B in *schedule*

Let's construct the spreadsheet graph for testing.

```
[1]: %run -i ./m269_digraph
```

```
spreadsheet = DiGraph()
for node in ("A2", "B2", "C2"):
    spreadsheet.add_node(node)
spreadsheet.add_edge("A2", "B2")
spreadsheet.add_edge("A2", "C2")
spreadsheet.add_edge("B2", "C2")
```

21.3.2 Algorithm and code

The key idea to obtain a topological sort is that the first node we visit (the first task we schedule) must not have incoming edges, otherwise it would have to come after some other node. Let's call the first visited node V.

V has no incoming edges but it may have an outgoing edge V → B. Since V was visited first, it will come before all other nodes. The ordering imposed by edge V → B is therefore satisfied.

If we remove all outgoing edges from V, to ‘discharge’ those order constraints, nodes that only depend on V will have no incoming edge anymore and can be visited next.

The algorithm thus proceeds by visiting and removing nodes with in-degree zero: those nodes depend on no other node and can be scheduled next. Here's the outline of what's known as

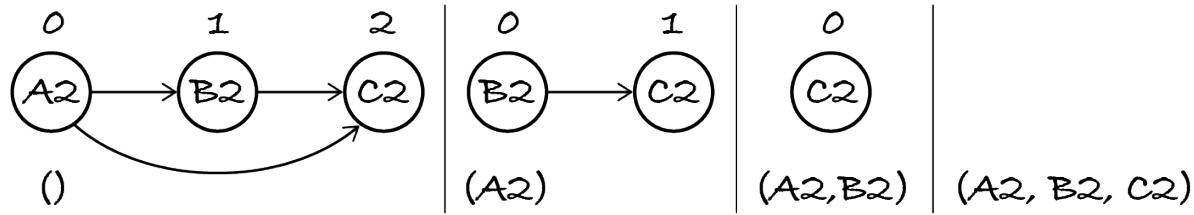
Kahn's algorithm:

Create an empty sequence. While there's a node with in-degree zero, remove it from the graph and append it to the sequence. When the while-loop ends, return the sequence.

It's a greedy algorithm: at each step it chooses one of the ‘best’ remaining nodes – those without incoming edges.

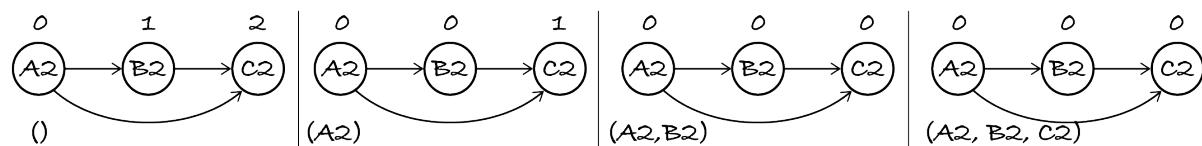
Let's see the algorithm in action on the example graph. The next figure shows from left to right how each iteration removes one node and appends it to the sequence. We start with a digraph and the empty sequence. We finish with the empty graph and a topological sort. The numbers next to the nodes are their in-degrees.

Figure 21.3.4



This version of the algorithm is not very good as it destroys the input graph. We can instead simulate the removal of nodes and how the in-degrees change. We store the initial in-degree of each node and simulate the removal of an edge $A \rightarrow B$ by decrementing the in-degree value of B . Here's the new version of the algorithm applied to the example graph. This version only changes the in-degree values associated to the nodes: it doesn't modify the graph. For example, the removal of A_2 and its edges is simulated by decrementing the in-degrees of B_2 and C_2 .

Figure 21.3.5



Kahn's algorithm can be reformulated as follows:

Create an empty sequence. Compute and store the in-degree of each node. Put all zero-degree nodes in a collection of nodes to visit. While the collection isn't empty, remove one node from it, append the node to the sequence and decrement the in-degree of the node's out-neighbours. If an out-neighbour's degree becomes zero, add that out-neighbour to the nodes to visit. When the while-loop ends, return the sequence.

The collection of nodes to visit can be a set, queue or stack. The latter is the most efficient in terms of memory and run-time.

```
[2]: %run -i ../m269_stack
```

```
[3]: # this code is also in m269_digraph.py
```

```
def topological_sort(graph: DiGraph) -> list:
    """Return a topological sort of graph.

    Preconditions: graph is acyclic
    Postconditions:
        - the output is a permutation of the graph's nodes
        - for every edge A -> B, node A appears before B in the output
    """
    schedule = []

    # compute the initial in-degrees
    indegree = dict()
```

(continues on next page)

(continued from previous page)

```

for node in graph.nodes():
    indegree[node] = 0
for edge in graph.edges():
    indegree[edge[1]] = indegree[edge[1]] + 1

# compute the nodes that can be visited first
to_visit = Stack()
for node in graph.nodes():
    if indegree[node] == 0:
        to_visit.push(node)

while to_visit.size() > 0:
    visited = to_visit.pop()
    schedule.append(visited)
    # simulate the removal of the visited node
    for neighbour in graph.neighbours(visited):
        indegree[neighbour] = indegree[neighbour] - 1
        if indegree[neighbour] == 0:
            to_visit.push(neighbour)
return schedule
    
```

[4]: topological_sort(spreadsheet)

[4]: ['A2', 'B2', 'C2']

21.3.3 Complexity

The algorithm always goes through the whole graph and adds all nodes to the output sequence, so there's no best- or worst-case scenario. The complexity can be broken down as follows:

- Go through the graph to construct the in-degree map: $\Theta(n + e)$.
- Do a linear search for the nodes with in-degree zero: $\Theta(n)$.
- Add each node to the `to_visit` set, remove it from the set and append it to the sequence: $\Theta(n)$.
- For each node, go through its out-neighbours: $\Theta(e)$, because visiting all neighbours of all nodes *goes through all edges*.

The complexity is given by the fastest-growing term: $\Theta(n + e)$.

21.3.4 Exercises

The following exercises show a different application of Kahn's algorithm and ask you to consider the efficiency of alternative algorithms.

Exercise 21.3.1

Alice remembers that the `DiGraph` class has a method to compute the in-degree. She simplifies the `topological_sort` code as follows. (The rest of the function remains the same.)

```
indegree = dict()
to_visit = set()
for node in graph.nodes():
    indegree[node] = graph.in_degree(node)
    if indegree[node] == 0:
        to_visit.add(node)
```

Is this more efficient than the original, unmodified code?

Hint Answer

Exercise 21.3.2

What happens if we ignore the preconditions of `topological_sort` and provide as input a cyclic digraph? Does the function stop with an error? Does it enter an infinite loop? If neither of those cases happen, what is the output?

Hint Answer

Exercise 21.3.3

1. Based on the previous exercise, write and test a function that checks if a digraph is cyclic.

```
[5]: from algoesup import test

def is_cyclic(graph: DiGraph) -> bool:
    """Return True if and only if the graph has a cycle."""
    pass

digraph = DiGraph()  # from Section 21.2.1
for node in "ABCDEF":
    digraph.add_node(node)
for edge in ("AB", "BC", "CA", "DE"):  # cycle A -> B -> C -> A
    digraph.add_edge(edge[0], edge[1])

is_cyclic_tests = [
    # case,           graph,           is cyclic?
    ('has cycle',   digraph,         True),
    ('no cycle',    spreadsheet,    False)
]

test(is_cyclic, is_cyclic_tests)
```

2. What's the worst-case scenario for your algorithm to decide whether a digraph is cyclic?
What's the worst-case complexity?

Hint Answer

Exercise 21.3.4

Looking at the tests above, Bob realises a digraph is cyclic if and only if it has a strongly connected component with two or more nodes. He adapts the algorithm for computing the *strongly connected components* as follows:

When computing the intersection of the *forward* and *backward* sets of nodes, check if the intersection's size is larger than 1. If so, immediately return true: the digraph is cyclic. Otherwise continue the algorithm as normal. Return false after the loop goes through all nodes: the digraph is acyclic because each component has one node only.

1. Explain why a cyclic digraph has a strongly connected component with more than one node.
2. What's the worst-case complexity of Bob's algorithm? Is it worth using his algorithm instead of Kahn's to check for cycles?

Hint Answer

21.4 State graphs

As I mentioned in [Section 17.1](#), directed graphs can represent states and transitions between states. The example I gave was the states of the board during a game of Noughts and Crosses (also known as Tic-tac-toe). The transitions between states are the player moves.

This section shows an example of a problem that can be easily solved if we model it with a state transition graph.

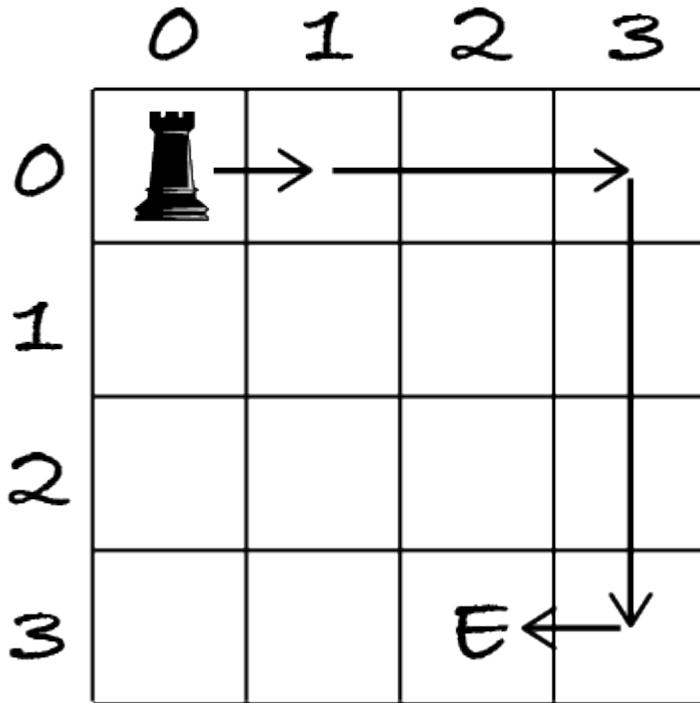
21.4.1 Problem

The problem to solve is similar to the one about a *knight moving on a chessboard*, but this time using a rook – a chess piece that moves only horizontally or vertically. We're given a rectangular board of squares, a start square and an end square. We want to find the fewest moves for the rook to go from the start to the end square, with the proviso that moves have to successively be 1, 2, 3, 1, 2, 3, 1, ... squares long until the end square is reached.

This means that the first move goes to an adjacent square, the second move jumps over one square, the third move jumps over two, the fourth move goes again to an adjacent square, etc. If there's a path (a sequence of moves) from the start to the end that stays within the board, then output the length of the path (the number of moves), otherwise output -1 .

The next figure shows a 4×4 board, with a rook symbol on the start square and an E marking the end square. For this input, the output should be four. A path of length four goes first one square right, then two squares right, then three squares down and finally one square left.

Figure 21.4.1



A path that first goes two squares to the right and then three down is shorter, but it's not a valid path because it doesn't start with a move of one square.

To check your understanding of the movement, find another shortest valid path from the start to the end, also in four moves.

Move one square down, then two squares down, then three to the right and finally one to the left.

As usual, I first construct some test cases. The inputs and output are exactly as for the knight moves problem: three input pairs of integers indicating the size of the board and the coordinates of the start and end squares, and one output integer indicating the shortest path length.

```
[1]: from algoesup import check_tests

rook_moves_tests = [
    # case,           size,   start,   end,   moves
    ('1x1 board',     (1, 1), (0, 0), (0, 0), 0),
    ('1 row, 2 cols', (1, 2), (0, 0), (0, 1), 1),
    ('start = end',   (3, 3), (1, 1), (1, 1), 0),
    ('figure example', (4, 4), (0, 0), (3, 2), 4),
    ('2 away',        (4, 4), (0, 0), (0, 2), -1)
]

check_tests(rook_moves_tests, [tuple, tuple, tuple, int])
```

OK: the test table passed the automatic checks.



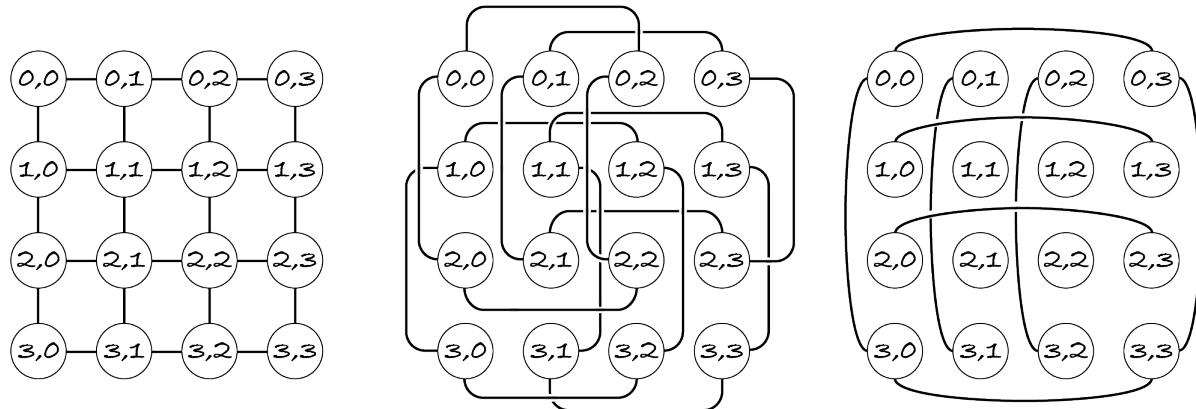
Info: This is a simplification of problem [Eternal Truths](#) from the 2004 Portuguese University Programming Contest.

21.4.2 Graph

The problem asks for the fewest moves. As for the knight moving on a chessboard, this seems to be a shortest path problem on an undirected graph, with one node per square. However, the distance of the move changes in each step, so the neighbours of each square are constantly changing. It seems we need three graphs instead of one.

Here they are for the example board above. In the left-hand graph, the edges connect the adjacent squares. In the middle graph, the edges connect nodes that are two squares away. In the right-hand graph, the edges connect nodes that are three squares away.

Figure 21.4.2



The algorithm would be a modified breadth-first search that is constantly switching between graphs, because the first, fourth, seventh, ... moves are done on the left-hand graph, the second, fifth, eighth, ... moves are done on the middle graph, the third, sixth, ninth, ... moves are done on the right-hand graph. This sounds too complicated and error-prone to me.

We need an approach that uses a single, unchanging graph, because all our graph algorithms work on such graphs, not on graphs where the edges are changing as the algorithm progresses.

The solution is to define a graph that represents the possible states of the rook. As breadth-first search explores the paths from the start node, it needs to know which square the rook is on and what move it can do next. The state of the rook is its current position and the distance of the next move.

For each square S we need three nodes (S, 1), (S, 2) and (S, 3) that represent the three possible states for when the rook is in that square: it can next move by one, two or three squares.

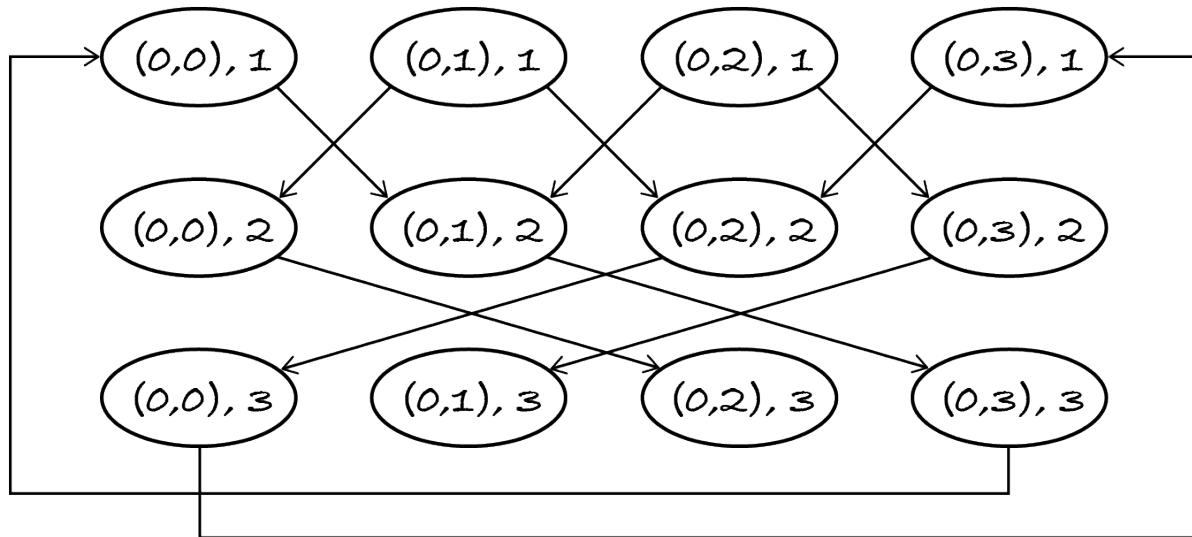
The graph has one edge (A, 1) —→ (B, 2) for each square B that is adjacent to square A. The edges state that the rook can move from any square to any adjacent square if the next move is by one square. Once it does the move, the rook is in a state where it next moves by two squares. Likewise there are edges

- (A, 2) —→ (B, 3) for each position B that is two squares away from A

- $(A, 3) \rightarrow (B, 1)$ for each position B that is three squares away from A.

Here's the state transition graph for a 1×4 board: a single row of four squares.

Figure 21.4.3



The layout of the edges shows when the rook can move left or right. The first move, from $(A, 1)$ to $(B, 2)$, goes to the square left or right of A, when possible. The second move, from $(A, 2)$ to $(B, 3)$, goes left or right by two squares when possible. Finally, the third move, from $(A, 3)$ to $(B, 1)$, is only possible from the left-most to the right-most square and vice versa.

Having constructed this graph, we apply BFS to find the shortest path from $(start, 1)$ to $(end, move)$ where $move$ can be any value. Once we reach the end square, we don't really care what the next move should be.

21.4.3 Code

First, I construct the state transition graph.

```
[2]: %run -i ../m269_digraph

def state_transitions(size: tuple) -> DiGraph:
    """Return the state transition graph for a board of the given size.

    Preconditions: size is a pair of positive integers, the number of rows and columns
    """
    rows = size[0]
    columns = size[1]
    states = DiGraph()
    # add nodes (S, 1), (S, 2), (S, 3) for every square S
    for row in range(rows):
        for col in range(columns):
            states.add_node((row, col, 1))
            states.add_node((row, col, 2))
            states.add_node((row, col, 3))

    for node in states.nodes:
        if node[0] == 0:
            if node[1] < columns - 1:
                states.add_edge(node, (node[0], node[1] + 1, 1))
                states.add_edge(node, (node[0], node[1] + 1, 2))
                states.add_edge(node, (node[0], node[1] + 1, 3))
            else:
                states.add_edge(node, (node[0], node[1], 1))
                states.add_edge(node, (node[0], node[1], 2))
                states.add_edge(node, (node[0], node[1], 3))
        elif node[0] == rows - 1:
            if node[1] > 0:
                states.add_edge(node, (node[0], node[1] - 1, 1))
                states.add_edge(node, (node[0], node[1] - 1, 2))
                states.add_edge(node, (node[0], node[1] - 1, 3))
            else:
                states.add_edge(node, (node[0], node[1], 1))
                states.add_edge(node, (node[0], node[1], 2))
                states.add_edge(node, (node[0], node[1], 3))
        else:
            if node[1] < columns - 1:
                states.add_edge(node, (node[0], node[1] + 1, 1))
                states.add_edge(node, (node[0], node[1] + 1, 2))
                states.add_edge(node, (node[0], node[1] + 1, 3))
            else:
                states.add_edge(node, (node[0], node[1], 1))
                states.add_edge(node, (node[0], node[1], 2))
                states.add_edge(node, (node[0], node[1], 3))
            if node[1] > 0:
                states.add_edge(node, (node[0], node[1] - 1, 1))
                states.add_edge(node, (node[0], node[1] - 1, 2))
                states.add_edge(node, (node[0], node[1] - 1, 3))
            else:
                states.add_edge(node, (node[0], node[1], 1))
                states.add_edge(node, (node[0], node[1], 2))
                states.add_edge(node, (node[0], node[1], 3))
```

(continues on next page)

(continued from previous page)

```

for column in range(columns):
    for move in (1, 2, 3):
        states.add_node(((row, column), move))

# add edges
for state in states.nodes():
    position = state[0]
    distance = state[1]

    row = position[0]
    column = position[1]

    next_distance = distance % 3 + 1 # 1 -> 2 -> 3 -> 1 -> ...

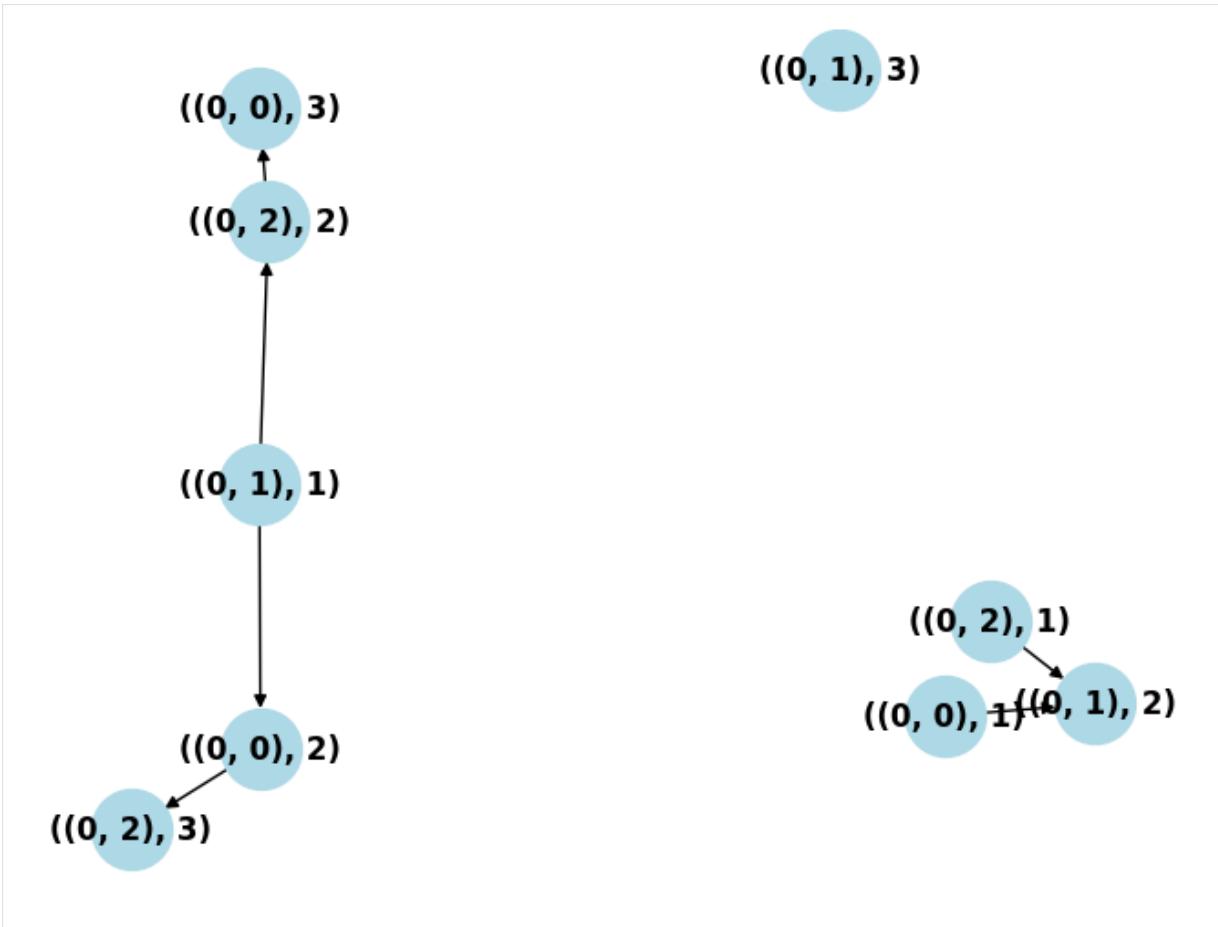
    # generate the 4 possible moves: up, left, down, right
    for move in (-distance, distance):
        # do vertical move if it stays within board
        if 0 <= row + move < rows:
            next_state = ((row + move, column), next_distance)
            states.add_edge(state, next_state)
        # do horizontal move if it stays within board
        if 0 <= column + move < columns:
            next_state = ((row, column + move), next_distance)
            states.add_edge(state, next_state)

return states

```

Before moving on, I test this function with a small 1×3 board. Remember that the graph layout is semi-random, so you may have to run the next cell a few times to get a more understandable drawing.

```
[3]: state_transitions((1, 3)).draw() # single row, three columns
```



You should be able to see, among other edges, $((0, 1), 1) \rightarrow ((0, 0), 2) \rightarrow ((0, 2), 3)$: the rook moves one square left from $(0, 1)$ to $(0, 0)$ and then two squares right to $(0, 2)$. Moving next by three squares would put the rook outside the board.

Next, I copy *the code* for the knight moves problem and modify it for this problem. I'll explain the changes after the code.

[4]: %run -i ./m269_queue

```

def rook_moves(size: tuple, start: tuple, end: tuple) -> int:
    """Return the least number of 1, 2, 3, 1, ... rook moves from
    start to end.

    Return -1 if end is not reachable from start.

    Preconditions:
    - size is a pair (rows, columns) with rows > 0 and columns > 0
    - start and end are pairs (r, c) with 0 <= r < rows and 0 <= c <
    columns
    """
    if end == start:
        return 0

```

(continues on next page)

(continued from previous page)

```

graph = state_transitions(size) # change 1

initial_state = (start, 1) # change 2
visited = {initial_state} # change 2
unprocessed = Queue()
for neighbour in graph.out_neighbours(initial_state): # change 2
    unprocessed.enqueue((neighbour, 1)) # change 4
while unprocessed.size() > 0:
    to_visit = unprocessed.dequeue()
    current = to_visit[0]
    length = to_visit[1]
    if current[0] == end: # change 3
        return length
    elif current not in visited:
        visited.add(current)
        for neighbour in graph.out_neighbours(current):
            unprocessed.enqueue((neighbour, length + 1)) # ↵
            ↵change 4
return -1
    
```

The changes were as follows, besides the trivial modifications to the header and docstring.

1. Replace the code that creates the graph for the knight's moves with a call to `state_transitions`.
2. The initial node is `(start, 1)` instead of `start` because nodes now represent states, not squares.
3. For the same reason, extract the square from the `current` node before comparing it to the `end` square.
4. Further simplify the BFS algorithm, which I could have already done for the knight moves problem. Instead of adding to the queue edge (A, B) with the length of the path to B , I just add B and the length, because the shortest path is not asked for, only its length.

Finally, let's run the code on the test table created at the start.

```
[5]: from algoesup import test

test(rook_moves, rook_moves_tests)

Testing rook_moves...
Tests finished: 5 passed (100%), 0 failed.
```

The moral of this and similar problems is:



Note: Instead of inventing a new graph algorithm, model the problem with a graph that allows you to apply or adapt a standard graph algorithm.

Nodes can represent anything, including places, tasks, events, states. Edges may be weighted and directed. This gives graphs a great modelling power. Once we represent the input as a graph, we can often solve the problem with a standard graph algorithm or some small adaptation of it, as done above, because many graph problems fall into one of a small number of categories: find a shortest path, a minimum spanning tree, a topological sort or the graph's components.

21.4.4 Complexity

The complexity of this kind of approach (transform the input into a graph and apply a graph algorithm) is $\Theta(n + e)$ to construct the graph plus whatever the complexity of the graph algorithm is. For this problem, the algorithm used is BFS so the overall complexity is $\Theta(n + e) + \Theta(n + e) = \Theta(n + e)$.

However, the complexity must be stated in terms of the input, not of the constructed graph. We must determine the number of nodes and edges in terms of the input variables, and restate the complexity in those terms.

Exercise 21.4.1

The main input variable is the size of the board: the number of rows r and the number of columns c .

1. How many nodes does the state transition graph have, in terms of r and c ? In other words, give an expression for n , using r and c .
2. How many edges does the state transition graph have at most? State e as an expression in terms of r and c .
3. Using the previous expressions, give the complexity $\Theta(n + e)$ in terms of r and c .

Hint Answer

21.5 Practice

This section is for you to practice solving problems by converting the input to a graph and then applying a graph algorithm. I present some guiding questions to help you figure out what kind of graph and what algorithm are needed to solve such problems.

I first recap the solution for the rook's moves problem of the previous section, but now following the guiding questions. Then I present a new problem for you to solve, following the same questions.

21.5.1 Rook's moves

The problem asked for the fewest vertical and horizontal rook moves from a start square to an end square, but the distance of each move varies. The output must be -1 if the end square can't be reached.

To obtain a solution, I had to find answers to the following questions.

- What graph problem is this problem most similar to?

The graph problems seen so far are: finding a shortest path, a shortest tour, a minimum spanning tree, all components, a topological sort and determining if a digraph is cyclic. Finding the least number of moves suggests a shortest path problem.

- What algorithm and graph are needed to solve that graph problem?

The shortest path problem can be solved with a breadth-first search if the graph isn't weighted and with Dijkstra's algorithm if it is, provided no weight is negative. Both algorithms work on directed and undirected graphs, so for this problem thinking about what algorithm is needed has unfortunately not helped restricting the kind of graph.

- What graph can be constructed from the input? In particular:

- What do the nodes represent? Are they places, states, tasks, events, ...?
- What do the edges represent? When is there an edge between nodes A and B?
- Are the edges directed? What do the directions represent?
- Are the edges weighted? Do weights represent distance, time, cost, ...?

The problem is about something (a rook) moving in a space (a board). This suggests nodes represent places (squares of the board) and edges connect nodes that can be reached from each other. However, each move depends on past moves, e.g. the rook can move to an adjacent square if it's the first move or the previous move was three squares. To know which move the rook can make at any point, we must know its state: the square it's on and how far it can move next. We therefore need a directed state transition graph. The answers to the above questions are as follows.

The graph is directed but not weighted. The nodes represent the rook's possible states and the edges the state transitions. The nodes are pairs (s, d) for each square s and distance $d = 1, 2, 3$. There's an edge $(A, d_A) \rightarrow (B, d_B)$ for each pair of squares A and B that are d_A squares away from each other. If $d_A = 3$, then $d_B = 1$, otherwise $d_B = d_A + 1$.

- Which standard graph algorithm can be used to produce the output? Which modifications, if any, are needed?

The graph algorithms seen so far are: BFS, DFS, Dijkstra's, Prim's and Kahn's algorithms. The graph isn't weighted, so we can use BFS to find the shortest path from the *start* square to the *end* square. The rook begins in state $(start, 1)$ but it can finish in one of three states: $(end, 1), (end, 2), (end, 3)$. We must modify the BFS algorithm to stop at any of those nodes. We must also modify it so that instead of returning the tree traversed from the start node it returns the length of the path to the target node first reached (or -1 if there's no path).

Now that you've seen how this set of questions led me to the solution of the rook's moves problem, here's a problem for you to solve, following the same questions.

21.5.2 Islands

This problem is about counting islands on a satellite image. The image was divided into small squares with the same area, say 10×10 metres. Each square is classified as water if most of its pixels are blue, otherwise as land. The result can be stored as a grid of Booleans but we'll use a more visual representation: letter L for land and space for water. The problem's input is a list of strings of the same length, like this:

```
[1]: image = [
    'LLL L ',
    'L L  L',
    'L LL L',
    ' LLL L'
]
```

Two land squares belong to the same island if they're vertically or horizontally adjacent.

The example has three islands:

1. a big island in the west, with 11 land squares surrounding a lake
2. a 3-square north–south island in the east
3. a single-square island in the north-east.

Given such a list of strings, we want to know how many islands there are.



Info: This is LeetCode problem [200](#) with a different input format.

Exercise 21.5.1

What graph problem is this problem most similar to?

Hint Answer

Exercise 21.5.2

What algorithm and graph are needed to solve that graph problem?

Hint Answer

Exercise 21.5.3

What graph can be constructed from the input? In particular:

- What do the nodes represent? Are they places, states, tasks, events, ...?
- What do the edges represent? When is there an edge between nodes A and B?
- Are the edges directed? What do the directions represent?
- Are the edges weighted? Do weights represent distance, time, cost, ...?

Hint Answer

Exercise 21.5.4

Which standard graph algorithm can be used to produce the output? Which modifications, if any, are needed?

Hint Answer

Exercise 21.5.5 (optional)

Implement the outlined solution, i.e. write code that takes a list of equal-length strings of Ls and spaces, and returns the number of islands. The code first constructs the graph you defined in the third exercise and then applies the algorithm you defined in the previous exercise.

21.6 Summary

In this chapter you've seen further practical problems on graphs:

- Compute the components of a graph to find the disconnected parts of a network.
- Compute a schedule that is compatible with the precedence given by the edges.
- Check if a digraph has a cycle.

You've also seen that finding a suitable graph representation can make problems easier to solve than designing bespoke algorithms.

The chapter introduced the following concepts and algorithms.

A **connected component** of an undirected graph is a largest possible subgraph of mutually reachable nodes. These components can be found with repeated traversals of any kind. Each traversal from a node A finds all other nodes in the same component as A. The complexity is linear in the size of the graph: $\Theta(n + e)$.

A **weakly connected component** of a digraph is a largest possible subgraph of mutually reachable nodes, if we ignore the direction of edges. These components can be found by computing the connected components of the undirected version of the digraph. The complexity is also $\Theta(n + e)$.

A **strongly connected component** of a digraph is a largest possible subgraph of mutually reachable nodes. These components can be found with repeated traversals of the original graph and its reverse. The worst-case complexity is $\Theta(n \times (n+e))$.

The **reverse graph** of a digraph has the same nodes and edges but with the directions reversed. The reverse graph represents the inverse relation of the original graph, e.g. 'A follows B' becomes 'B is followed by A'. The reverse graph is computed in $\Theta(n + e)$ time.

A **topological sort** of a directed acyclic graph (DAG) is a permutation of its nodes so that for every edge $A \rightarrow B$, node A comes before node B in the permutation. A DAG has one or more topological sorts. Cyclic digraphs have no topological sort.

Kahn's algorithm computes a topological sort in a greedy fashion. In each iteration it (virtually) removes one node with in-degree zero from the digraph and appends it to an initially empty sequence. The complexity is $\Theta(n + e)$.

If the digraph has a cycle, Kahn's algorithm returns a sequence without all the graph's nodes, thus making it easy to detect cycles in digraphs.

CHAPTER 22

BACKTRACKING

In Chapter 11 you learned about *exhaustive search*, also called brute-force search or generate and test in M269. It iteratively generates candidates and tests them to check if they are solutions. This usually leads to generating far more candidates than there are solutions. There are some techniques to prune the search space, i.e. to generate fewer candidates, but they depend on the problem.

This chapter introduces backtracking, a search technique that is elegant and concise (because it's recursive), efficient (because it can prune the search space substantially) and systematic (because it always follows the same template). The technique only applies to problems where each solution (and therefore each candidate) is a collection of items, because backtracking relies on generating each candidate incrementally, one item at a time.

Section 1 of this chapter introduces a recursive exhaustive search for sequences of items that satisfy some constraints. Section 2 turns exhaustive search into a backtracking algorithm by making a small change that prunes the search space. Section 3 shows a typical example of applying backtracking: solving a puzzle.

Sections 4 and 5 show how backtracking can also solve optimisation problems, where one best among several possible solutions is asked for, using the travelling salesman problem as an example.

Finally, Sections 6 and 7 explain how to adapt backtracking to solve problems that are about sets of items instead of sequences of items, using the 0/1 knapsack problem as an example.



Info: Some authors use the term ‘exhaustive search’ to also include backtracking, but the latter in fact avoids searching all candidates.

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes - you will learn about backtracking, which is more efficient than

brute-force search.

- Write readable, tested, documented and efficient Python code – this chapter introduces code templates for backtracking that you can adapt to solve various problems.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

22.1 Generate sequences

Backtracking can solve many *constraint satisfaction and optimisation problems*. In M269 we restrict backtracking to problems on sets of items or on sequences of unique items, i.e. without duplicates. We will first look at constraint satisfaction problems on sequences. Here's one, admittedly contrived.

Given an integer $n > 2$, obtain all permutations of $1, \dots, n$ such that:

- the first and last numbers are at least $n / 2$ apart (range constraint)
- the sequence starts with an odd number and then alternates even and odd numbers (parity constraint).

For $n = 3$, only permutations $(1, 2, 3)$ and $(3, 2, 1)$ satisfy both constraints. Permutations like $(2, 1, 3)$ satisfy neither:

- the difference between the first and last numbers is 1, but it should be at least 1.5
- the permutation starts with an even number and has consecutive odd numbers.

I first solve the problem with a recursive exhaustive search, because backtracking is based on that.

A *brute-force search* for the solutions to the above problem generates all possible permutations and tests each permutation against both constraints, to decide if it's a solution.

To test the range constraint I compute the difference between the first and last numbers with Python function `abs`, introduced in [Section 18.2](#).

```
[1]: def satisfies_range(numbers: list, n: int) -> bool:  
    """Check if first and last of numbers are at least n/2 apart.  
  
    Preconditions:  
    - numbers is a list of integers; len(numbers) >= 2  
    - n > 2  
    """  
    return abs(numbers[0] - numbers[-1]) >= n / 2
```

To test the parity constraint, I must verify there's an odd number at index 0, an even number at index 1, an odd number at index 2, etc. The general rule is that each number and its index must have different parities. If there are no numbers (the list is empty) then the constraint is satisfied.

```
[2]: def satisfies_parity(numbers: list) -> bool:
    """Check if numbers is an odd, even, odd, ... sequence.

    Preconditions: numbers is a list of integers
    """
    for index in range(len(numbers)):
        if index % 2 == numbers[index] % 2:
            return False
    return True
```

The test for whether a permutation candidate is a solution is simply:

```
[3]: def is_solution(permuation: list, n: int) -> bool:
    """Check if permuation satisfies the range and parity-
    →constraints.

    Preconditions: n > 2 and permuation is a rearrangement of 1, ...
    →, n
    """
    return satisfies_range(permuation, n) and satisfies_
    →parity(permuation)
```

Now all that remains is to generate the permutations. This could be done iteratively with Python's `permutations` function introduced in [Section 11.4.4](#). However, backtracking requires each permutation to be generated one item at a time. (I'll explain why in Section 2.) We need a different way of generating permutations.

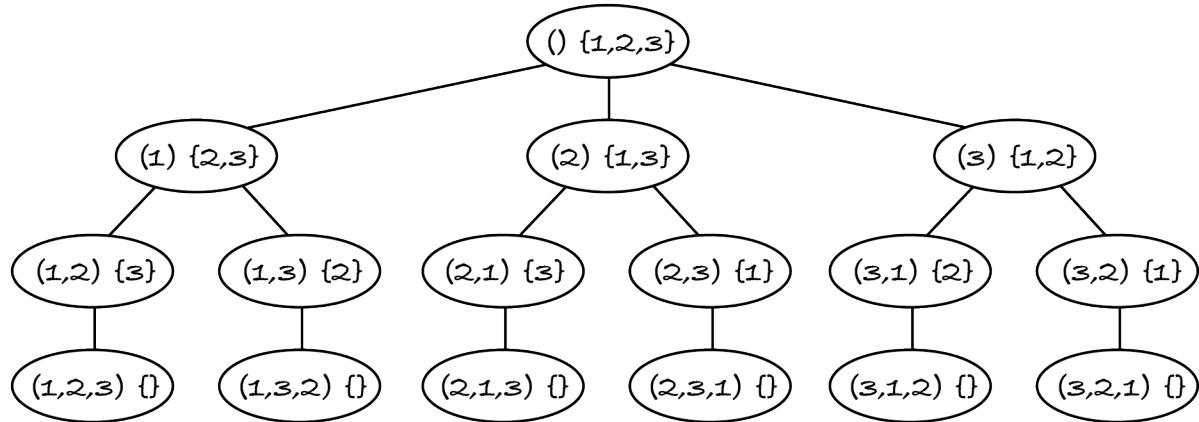
22.1.1 Recursive generation

To generate *one* permutation with pencil and paper, we could start with an empty sequence and take the numbers one by one, in any order, from the set $\{1, \dots, n\}$, appending each one to the sequence. When the set becomes empty, the sequence is a permutation of the numbers from 1 to n .

To generate *all* permutations, we need to systematically go back to an earlier choice (e.g. when we picked the first number in the permutation) and make a different one.

The following tree shows the decision process for $n = 3$. Each node represents the sequence created so far and the available set of choices. Initially the sequence is empty and all numbers are available. The children of a node are all the possible ways of extending the sequence in that node. When the set is empty, the sequence can't be further extended. Nodes with the empty set are the leaves of the tree and contain the permutations.

Figure 22.1.1



The sequences in the leaves are the **complete candidates**, which for this problem are the permutations. The sequences in the other nodes are the **partial candidates**. Each set in a node is the **extensions** for that node's candidate. A candidate is complete when it has no extensions. A **solution** is a complete or partial candidate that satisfies the constraints. For this problem, all solutions are complete candidates (permutations). Because of the constraints imposed, not every complete candidate is a solution.

Since the candidates can be organised in a tree, all we need to generate them is a recursive tree-traversal algorithm. The *algorithms* in Chapter 16 went through an existing tree. It would be a waste of time and memory to create the whole tree in advance, with the partial candidates and their extensions, as we're only interested in those complete candidates that are solutions. A better approach is to create the nodes as they're visited. In fact, we don't need to create node objects with pointers to children: we only need the content of nodes (the candidates and their extensions), which is much simpler and efficient.

The following is a recursive exhaustive search. It generates all partial and complete candidates and their extensions, and tests complete candidates (the permutations) against the constraints. If the permutation is a solution then it's appended to a sequence of solutions, so that we keep the solutions in the order they're found. I've also added some print statements to follow what the search does.

```
[4]: def extend(candidate: list, extensions: set, n: int, solutions:_
           <list>) -> None:
    """Add to solutions all valid permutations that extend candidate.

    Preconditions: n > 2 and
    - candidate is a list of integers between 1 and n
    - extensions is a set of integers between 1 and n
    - candidate and extensions have no integer in common
    """
    print("Visiting node", candidate, extensions)
    if len(extensions) == 0: # leaf node: candidate is complete
        print("Testing candidate", candidate)
        if is_solution(candidate, n):
            solutions.append(candidate)
    else: # create and visit children nodes
        for item in extensions:
```

(continues on next page)

(continued from previous page)

```
extend(candidate + [item], extensions - {item}, n,
→solutions)
```

Like all recursive algorithms, it has a base case (there are no extensions) and a reduction step (remove one item from the extensions) to make progress towards the base case.

Being a tree-traversal function, I must call it on the root node: the empty candidate and the full set of extensions. I must also initialise the solutions sequence.

```
[5]: def valid_permutations(n: int) -> list:
    """Return all valid permutations of 1, ..., n in the order
    →generated."""
    candidate = []
    extensions = set(range(1, n + 1)) # {1, ..., n}
    solutions = []
    extend(candidate, extensions, n, solutions)
    return solutions

print("Solutions:", valid_permutations(3))

Visiting node [] {1, 2, 3}
Visiting node [1] {2, 3}
Visiting node [1, 2] {3}
Visiting node [1, 2, 3] set()
Testing candidate [1, 2, 3]
Visiting node [1, 3] {2}
Visiting node [1, 3, 2] set()
Testing candidate [1, 3, 2]
Visiting node [2] {1, 3}
Visiting node [2, 1] {3}
Visiting node [2, 1, 3] set()
Testing candidate [2, 1, 3]
Visiting node [2, 3] {1}
Visiting node [2, 3, 1] set()
Testing candidate [2, 3, 1]
Visiting node [3] {1, 2}
Visiting node [3, 1] {2}
Visiting node [3, 1, 2] set()
Testing candidate [3, 1, 2]
Visiting node [3, 2] {1}
Visiting node [3, 2, 1] set()
Testing candidate [3, 2, 1]
Solutions: [[1, 2, 3], [3, 2, 1]]
```

As you can see, the algorithm tests all $3! = 6$ permutations of numbers 1 to 3, but only two of them are solutions.

If you follow the nodes visited with your finger on the tree diagram, you see that the algorithm is

doing a pre-order traversal of the tree. After visiting a leaf and testing the complete candidate in it, the algorithm ‘unwinds’ (because leaves have no children) to the last node with yet unvisited subtrees and traverses the next subtree.

For example, if you look at the printed output and at the tree diagram, after producing permutation [1, 3, 2], there are no further subtrees to explore for partial candidate [1]. The execution of the algorithm is back in the for-loop of the call on the root node `extend([], {1, 2, 3}, 3, solutions)` and goes into the next iteration, with `item` being 2. The next recursive call is `extend([2], {1, 3}, 3, solutions)`, which starts traversing the middle subtree.

At this point you may be rightly thinking that this recursive brute-force search is less efficient than an iterative one, because it also generates and visits all partial candidates as the initially empty candidate is extended one item at a time. For $n = 3$, there are 10 partial and only 6 complete candidates.

One advantage of the incremental generation approach is that it can also solve problems where the solution sequences don’t have all items, i.e. where some partial candidates are solutions too.

22.1.2 Accept partial candidates

Let’s change the problem so that solutions can be sequences with only some of the numbers from 1 to n , as long as they satisfy both constraints. All permutations satisfying the range and parity constraints are still solutions, but shorter sequences may be solutions too. For example, for $n = 4$, (1, 2, 3, 4) is the only permutation that is a solution, but sequences (1, 2, 3) and (3, 2, 1) are solutions too: they each have a difference of at least $4 / 2 = 2$ between the first and last numbers and they alternate odd and even numbers.

To solve this problem I must make two changes. First, the tree traversal must check every candidate, not just the complete candidates. (I don’t repeat the docstring and print statements.)

```
[6]: def extend(candidate: list, extensions: set, n: int, solutions: list) -> None: # noqa: D103
    if is_solution(candidate, n):
        solutions.append(candidate)
    for item in extensions:
        extend(candidate + [item], extensions - {item}, n, solutions)
```

The change I made was removing the if-else statement that checked if the candidate is complete. The for-loop won’t do anything for complete candidates because they have no extensions.

The second change is to the function that tests the range constraint, which requires the sequence to have at least two numbers. Previously, we could assume that as part of the preconditions, because only complete candidates (permutations of $n > 2$) were tested. Now that we test all partial candidates, we must explicitly check they have at least two numbers.

```
[7]: def satisfies_range(numbers: list, n: int) -> bool:
    """Check if first and last of numbers are at least n/2 apart.

    Preconditions: numbers is a list of integers; n > 2
```

(continues on next page)

(continued from previous page)

```
"""
return len(numbers) > 1 and abs(numbers[0] - numbers[-1]) >= n / 2
```

Let's confirm that we now find more sequences for $n = 4$.

```
[8]: valid_permutations(4)
```

```
[8]: [[1, 2, 3], [1, 2, 3, 4], [1, 4], [1, 4, 3], [3, 2, 1], [3, 4, 1]]]
```

To sum up, a constraint satisfaction problem on sequences with unique items can be solved with a brute-force search that generates all candidates in a recursive and incremental way, starting from the empty sequence and extending it by one item at a time. Depending on the problem, all candidates or only the complete ones are tested against the constraints.

The next section shows the main advantage of incremental generation: a simple change will make the search much more efficient.

22.2 Prune the search space

Backtracking takes the brute-force search of the previous section and adds a simple but powerful idea: since the candidates are generated incrementally, one item at a time, stop extending a candidate as soon as it's clear it won't lead to a solution. This substantially reduces the number of candidates generated, making backtracking much more efficient than brute-force.

Let's see backtracking in action on the problem of the previous section: find all sequences of non-repeated numbers, taken from 1 to $n > 2$, such that

1. the first and last numbers are at least $n / 2$ apart (range)
2. the numbers are odd, even, odd, even, ... (parity).

The sequences don't have to be permutations of 1 to n : they can include only some of the n numbers.

Here again is the code that checks these constraints.

```
[1]: def satisfies_range(candidate: list, n: int) -> bool:
    """Check if first and last numbers in candidate are at least n/2 apart.

    Preconditions: candidate is a list of integers; n > 2
    """
    return len(candidate) > 1 and abs(candidate[0] - candidate[-1]) >= n / 2

def satisfies_parity(candidate: list) -> bool:
    """Check if candidate is an odd, even, odd, ... sequence.
```

(continues on next page)

(continued from previous page)

```

Preconditions: candidate is a list of integers
"""
for index in range(len(candidate)):
    if index % 2 == candidate[index] % 2:
        return False
return True

```

22.2.1 Local and global constraints

The key insight that enables pruning the search space is that the two constraints for this problem are of different nature. The range constraint involves the first and last numbers of the sequence and therefore can only be checked on the whole sequence: it's a **global constraint**. The second constraint is about the parity of each number, independently of the other numbers: it's a **local constraint**.

If a partial candidate P doesn't satisfy a global constraint, an extension of P may satisfy it because of the added items. For example, for $n = 3$ the sequence (1, 2) doesn't satisfy the range constraint but (1, 2, 3) does. We therefore must keep extending a candidate that fails a global constraint.

However, if partial candidate P doesn't satisfy a local constraint, neither does any candidate C that extends P because the item in P that breaks the local constraint is also in C. For example, if P starts with an even number, or has two consecutive odd numbers, then so does any extension of P. This means that there's no point in extending a partial candidate that violates a local constraint: it won't lead to a solution.

If a candidate fails the local constraints, a **backtracking algorithm** goes immediately back (hence its name) to a previous partial candidate and tries a different way to extend it.

In terms of tree traversal, if a node has a candidate that fails the local constraints, a backtracking algorithm doesn't traverse the subtree rooted at that node: it instead goes back to the node's parent. From there it starts traversing the next sibling subtree. If there's no sibling, the algorithm backtracks to the grandparent, and so on.

The changes to the previous recursive generate-and-test algorithm are minor: I simply add a new base case. If the candidate fails the local constraints, the algorithm returns (backtracks) immediately instead of recursively extending the candidate.

```

[2]: def extend(candidate: list, extensions: set, n: int, solutions:_
           →list) -> None:
    """
    Add to solutions all valid permutations that extend candidate.

    Preconditions: n > 2 and
    - candidate is a list of integers between 1 and n
    - extensions is a set of integers between 1 and n
    - candidate and extensions have no integer in common
    """

```

(continues on next page)

(continued from previous page)

```

print("Visiting node", candidate, extensions)
# base case 1: backtrack
if not satisfies_parity(candidate):
    return
# base case 2: candidate is solution
# local constraint is satisfied, so only check global constraint
if satisfies_range(candidate, n):
    solutions.append(candidate)
for item in extensions:
    extend(candidate + [item], extensions - {item}, n, solutions)
    
```

The main function remains the same.

```

[3]: def valid_permutations(n: int) -> list:
    """Return all valid permutations of 1, ..., n in the order
    generated."""
    candidate = []
    extensions = set(range(1, n + 1))  # {1, ..., n}
    solutions = []
    extend(candidate, extensions, n, solutions)
    return solutions

print("Solutions:", valid_permutations(3))

```

Visiting node [] {1, 2, 3}
 Visiting node [1] {2, 3}
 Visiting node [1, 2] {3}
 Visiting node [1, 2, 3] set()
 Visiting node [1, 3] {2}
 Visiting node [2] {1, 3}
 Visiting node [3] {1, 2}
 Visiting node [3, 1] {2}
 Visiting node [3, 2] {1}
 Visiting node [3, 2, 1] set()
 Solutions: [[1, 2, 3], [3, 2, 1]]

Now only 10 of the 16 nodes of the full tree are visited. As you can see, sequences (1, 3), (2) and (3, 1) aren't further extended because they break the parity constraint.

As suggested in [Chapter 15](#), we can further reduce the search space by not even generating those sequences, since they will be rejected.

22.2.2 Avoid visits

The algorithm currently first extends a candidate and then checks whether it should backtrack. Since the local constraint applies to each number individually, we can check the parity of the chosen extension number *before* appending it to the sequence. In other words, instead of creating and visiting a node and then backtracking if needed, we avoid creating the node in the first place. This further prunes the search space.

Here's the new version, without repeating the docstring. The check for the local parity constraint moves to the for-loop: if the chosen number can extend the current sequence, it will.

```
[4]: def extend(candidate: list, extensions: set, n: int, solutions: list) -> None: # noqa: D103
    print("Visiting node", candidate, extensions)
    if satisfies_range(candidate, n):
        solutions.append(candidate)
    for item in extensions:
        if can_extend(item, candidate): # added line
            extend(candidate + [item], extensions - {item}, n, solutions)
```

Instead of an explicit base case that goes back to a previous candidate if the current one doesn't satisfy the local constraint, the new version avoids extending candidates with items that fail the local constraint.

The new auxiliary function checks the local constraint on a single number instead of a sequence and hence is simpler than `satisfies_parity`.

```
[5]: def can_extend(item: int, candidate: list) -> bool:
    """Check if extending candidate with item can lead to a solution.
    """
    # the number and the index where it will be must have different parity
    return item % 2 != len(candidate) % 2

valid_permutations(3)

Visiting node []
Visiting node [1]
Visiting node [1, 2]
Visiting node [1, 2, 3]
Visiting node [1, 2, 3] set()
Visiting node [3]
Visiting node [3, 2]
Visiting node [3, 2, 1]
Visiting node [3, 2, 1] set()

[[1, 2, 3], [3, 2, 1]]
```

As expected, no partial candidate with consecutive numbers of the same parity or starting with an even number is generated. Now only 7 of the 16 nodes are created and visited. The search space has more than halved.

To emphasise how much more efficient backtracking can be, consider $n = 10$ and only complete candidates, i.e. permutations of the ten numbers. There are $5 \times 9! \approx 1.8$ million permutations starting with one of the five even numbers (2, 4, 6, 8, 10) followed by a permutation of the other nine numbers. There are further $5 \times 4 \times 8! \approx 800$ thousand permutations starting with two consecutive odd numbers, followed by a permutation of the other eight numbers. Backtracking won't generate these 2.6 million permutations and thousands of other ones with consecutive numbers of the same parity, whereas brute-force search generates all $10! \approx 3.6$ million permutations.

To sum up, backtracking can efficiently find sequences of items subject to global and local constraints because it generates candidates incrementally. Global constraints are checked on all candidate sequences or only on the complete ones, depending on the problem. Local constraints are checked for each extension being considered, to avoid generating candidates that won't lead to solutions. This usually prunes vast parts of the search space. If a problem has no local constraints then backtracking becomes brute-force search because all candidates have to be generated.

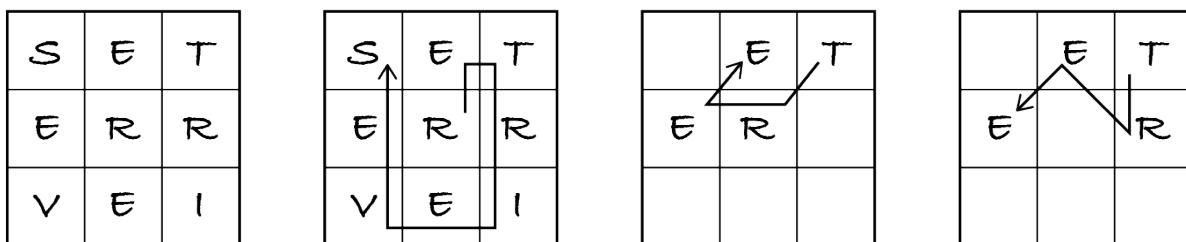
The next section shows a typical problem that can be solved with backtracking.

22.3 Trackword

Trackword is a puzzle to find as many words with three to nine letters as we can in a 3×3 grid of letters. We can start in any square and move to any adjacent square, without visiting any square twice. The grid always contains a 9-letter word.

Here's an example created by a [fan site](#). The figure shows the grid, the path for the 9-letter word 'retrieves' and two paths for the word 'tree'. The grid also includes words ire, set, tee, sever, reset, reverse, serve, etc.

Figure 22.3.1



For this and other word problems we need a list of valid words. The notebooks folder for this chapter includes the public-domain Enable list, which contains long and inflected words (past tense, plural, etc.). The words in the list are in lowercase, so the grid will be in lowercase too.



Info: Enable and other word lists are available from [The National Puzzler's League](#).

The following code reads words with three to nine letters from the file. It uses Python constructs outside the scope of M269.

```
[1]: vocabulary = set()
with open("enable1.txt") as file:
    for line in file:
        word = line.strip()
        if 3 <= len(word) <= 9:
            vocabulary.add(word)
```

Let's solve this problem with backtracking. I will ask you to think along with 'stop and think' lines.

22.3.1 The problem

The first thing to ponder is whether this problem is adequate for backtracking:

1. Is this a constraint satisfaction problem on sequences of unique items? If so, what are the sequences, why are items unique and what are the other constraints?

Yes, it is. We want to find all sequences of squares subject to these constraints: the length of the sequence is three to nine, the sequence is a valid path and the letters on the squares form a word. No square can be visited twice so the sequence has no duplicate squares.

2. Which of the constraints are global and which are local?

The global constraints are that the sequence forms a word and the word comprises three to nine letters. The local constraint is that each square in the sequence (except the first one) must be adjacent to the previous one.



Note: Backtracking is usually applicable to problems involving paths on 2D grids.

22.3.2 Candidates and extensions

Let's start solving this problem. The core of backtracking is to extend a candidate sequence with one item at a time, taken from an initial set with all possible extensions. So the obvious starting questions to solve a problem are:

1. What are the candidates? What do they represent?
2. What are the items in the candidates? What do items represent?

-
1. There are only two options for what a candidate can be: the string of letters or the path of squares visited so far. As the example at the start shows, there may be several paths associated to the same string. If a candidate were a string, we couldn't determine which squares can be visited next. The candidates are therefore paths: each one is a sequence of squares visited.

2. Each item represents a square of the grid.

Once we have thought what each item represents in the problem, we need to think how to represent each item in the solution.

3. What data structure should we use for each item?
 4. What is the initial set of all possible extensions? How can it be generated?

3. We can store each square as a pair of integers with the square's row-column coordinates.
 4. The initial extensions are all 9 squares of the grid. They can be generated with a nested loop iterating over all rows and columns of the grid.

At this point we know the problem's inputs and the initial extensions, which is enough to write the main function. I always pass the problem's inputs to the backtracking function because it may need them to check the global and local constraints.

```
[2]: def trackword(grid: list, valid: set) -> list:  
    """Return all words found in the grid, in the order generated.  
  
    Preconditions:  
    - grid is a 3 by 3 table of lowercase letters  
    - valid is a set of strings of allowed words  
    """  
  
    path = [] # the initial candidate  
    squares = set() # the extensions  
    for row in range(3):  
        for column in range(3):  
            squares.add((row, column))  
    solutions = []  
    extend(path, squares, grid, valid, solutions)  
    return solutions
```

The backtracking algorithm needs an extra if-statement if only complete candidates can be solutions, so the next thing to consider is:

5. Which candidates are solutions: only complete candidates or both complete and partial candidates?

The words in the grid can be of any length from 3 to 9, so both partial and complete paths may be solutions.

We can now write the backtracking algorithm, because it's boilerplate code that calls two auxiliary functions, passing them the problem's input in case they need it.

```
[3]: def extend(path: list, squares: set, grid: list, valid: set, ←  
→solutions: list) -> None:  
    """Extend the path with the squares. Add valid words to  
→solutions."""
```

(continues on next page)

(continued from previous page)

```
if is_word(path, grid, valid): # check the global constraints
    solutions.append(path)
```

As I write the last line I stop in my tracks. The candidates are paths in the grid, but the problem asks for the words, so I must convert each path of squares to the corresponding string of letters.

```
[4]: def path_to_string(path: list, grid: list) -> str:
    """Return the sequence of letters visited by the path in the
    grid."""
    string = ""
    for square in path:
        string = string + grid[square[0]][square[1]]
    return string
```

Now I can write the complete backtracking algorithm.

```
[5]: def extend(path: list, squares: set, grid: list, valid: set,
             solutions: list) -> None:
    """Extend the path with the squares. Add valid words to
    solutions."""
    if is_word(path, grid, valid):
        solutions.append(path_to_string(path, grid))
    for square in squares:
        if can_extend(square, path, grid, valid):
            extend(path + [square], squares - {square}, grid, valid,
                   solutions)
```

Finally, let's implement the auxiliary functions.

22.3.3 The constraints

The global constraints on a path are: it has three to nine letters and they form a valid word. Since only words with three to nine letters were read from the file, I can simply check if the path forms a word.

```
[6]: def is_word(path: list, grid: list, valid: set) -> bool:
    """Check if the letters in the path form a valid word."""
    return path_to_string(path, grid) in valid
```

As for the local constraint, we must check if the next square is adjacent to the last square in the path so far. If the path is empty, it can be extended with any square.

```
[7]: def can_extend(square: tuple, path: list, grid: list, valid: set) ->
         bool:
    """Check if square is adjacent to the last square of path."""
    if path == []:
        return True
```

(continues on next page)

(continued from previous page)

```

last_square = path[-1]
last_row = last_square[0]
last_column = last_square[1]
row = square[0]
column = square[1]
return abs(row - last_row) < 2 and abs(column - last_column) < 2
    
```

That's it! Let's check it works. There are probably many words in this grid, so let's just see a few of them.

```
[8]: words = trackword(["set", "err", "vei"], vocabulary)
print("Total paths:", len(words))
print("Found 9-letter word?", "retrieves" in words)
print("First 10:", words[:10])

```

```

Total paths: 156
Found 9-letter word? True
First 10: ['ere', 'err', 'errs', 'err', 'ere', 'ers', 'ere', 'eerie',
 ↵ 'eerier', 'res']
    
```

Some words occur repeatedly in the output list because there are various paths to obtain them. We can easily compute how many unique words there are:

```
[9]: len(set(words)) # convert to set to remove duplicates

```

```
[9]: 59
    
```

Exercise 22.3.1 (optional)

Looking at the code again, two improvements come to my mind. First, return a set instead of a sequence of words, to avoid duplicates. Second, remove unnecessary parameters. For example, function `can_extend` doesn't need the `grid` and `valid` arguments.

Copy all functions to the cell below, make the suggested changes and any others you wish and run the code to check it still finds 59 words in the grid.

```
[10]: # copy the functions to here

len(trackword(["set", "err", "vei"], vocabulary)) == 59
    
```

Exercise 22.3.2 (optional)

Implement a more substantial change: represent each candidate as a sequence–string pair. This is a space–time tradeoff to avoid repeatedly converting a path to a string. When extending a candidate, append a square to the sequence and the corresponding letter to the string.

22.3.4 Template

In general, to solve a problem with backtracking, first think of what the items, candidates and extensions represent and which are the global and local constraints. Then follow this backtracking solution template, replacing the generic function and variable names and docstrings with problem-specific ones, and removing unnecessary parameters.

```
def problem(instance: object) -> list:
    """Return all solutions for the problem instance, in the order
    generated."""
    candidate = []
    extensions = ...
    solutions = []
    extend(candidate, extensions, instance, solutions)
    return solutions

def extend(candidate: list, extensions: set, instance: object,
          solutions: list) -> None:
    """Add to solutions all extensions of candidate that solve the
    problem instance."""
    # remove next line if partial candidates can be solutions
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            solutions.append(candidate)
    for item in extensions:
        if can_extend(item, candidate, instance):
            extend(candidate + [item], extensions - {item}, instance,
                   solutions)

def satisfies_global(candidate: list, instance: object) -> bool:
    """Check if candidate satisfies the global constraints."""
    return ...

def can_extend(item: object, candidate: list, instance: object) ->
    bool:
    """Check if item may extend candidate towards a solution."""
    return ...
```

22.4 Optimise

I've shown you how to solve constraint satisfaction problems on sequences with backtracking. It shouldn't be a surprise that backtracking can also solve optimisation problems. Instead of collecting all solutions in a list or set, the algorithm keeps only the best solution found so far. Let's see an example.

22.4.1 The problem

I take the earlier problem of finding sequences of numbers from 1 to n that satisfy range and parity constraints, and add an optimisation criterion: we want a sequence with a lowest sum. For example, for $n = 4$, the best sequence is (1, 4) because its sum 5 is lowest among all other sequences that satisfy both constraints.

An optimisation problem asks to find a solution that minimises or maximises some value. We thus need a function that calculates the value of a candidate. For this problem, it's simply the sum of the numbers in the sequence.

```
[1]: def value(numbers: list) -> int:
    """Return the value of the candidate: the sum of the numbers."""
    total = 0
    for number in numbers:
        total = total + number
    return total
```

With respect to the earlier solution for the constraint satisfaction problem, I need to make two changes. First, instead of appending each solution found to a sequence, the backtracking algorithm must receive the current best solution as input and update it when it finds a better solution. Second, the main function must compute some initial best solution and pass it to the backtracking function. Let's do these changes in turn.

22.4.2 Keep the best

The backtracking algorithm must compare each found solution to the current best and update the latter if the new solution is better. If it's a minimisation problem, better means having a lower value, otherwise better means higher.

To avoid recomputing the best solution's value every time it's compared to a new solution, I will trade space for time and keep a solution–value pair instead of just the best solution. Usually I represent a pair with a tuple, but to be able to update the pair, I use a list of length two with *constants to name the indices*.

```
[2]: SOLUTION = 0
VALUE = 1
```

I could instead have defined a class with two data fields, similar to the node classes for *linked lists* and *binary trees*. I'll leave that as an optional exercise, if you wish to practise using classes.

The backtracking function is largely the same as before. The `solutions` parameter is replaced by the best solution–value pair, which is updated when the current candidate is a better solution.

I add some print statements to later follow the algorithm's actions.

```
[3]: def extend(candidate: list, extensions: set, n: int, best: list) ->
    None:
    """Update best if needed, and extend candidate."""
    print("Visiting node", candidate, extensions)
```

(continues on next page)

(continued from previous page)

```

if satisfies_range(candidate, n):
    candidate_value = value(candidate)
    if candidate_value < best[VALUE]:
        print("New best", candidate, "with value", candidate_
→value)
        best[SOLUTION] = candidate
        best[VALUE] = candidate_value
    for item in extensions:
        if can_extend(item, candidate):
            extend(candidate + [item], extensions - {item}, n, best)

```

The current best solution must be initialised before starting the search. Since the search updates the current best every time a better one is found, the initial best can be *any* solution, preferably one that is easy to construct.

For this problem, sequence $(1, 2, \dots, n)$ is a solution: it satisfies both constraints.

```
[4]: def best_sequence(n: int) -> list:
    """Return a lowest sum sequence that satisfies both constraints."""
    candidate = []
    extensions = set(range(1, n + 1))
    solution = list(range(1, n + 1))
    best = [solution, value(solution)]
    extend(candidate, extensions, n, best)
    return best
```

The constraint checking functions are as before.

```
[5]: def satisfies_range(candidate: list, n: int) -> bool:
    """Check if first and last numbers in candidate are at least n/2_
→apart.

    Preconditions: candidate is a list of integers; n > 2
    """
    return len(candidate) > 1 and abs(candidate[0] - candidate[-1]) >
→= n / 2

def can_extend(item: int, candidate: list) -> bool:
    """Check if extending candidate with item can lead to a solution.
    """
    # the number and the index where it will be put must have_
→different parity
    return item % 2 != len(candidate) % 2
```

(continues on next page)

(continued from previous page)

```
best_sequence(4)

Visiting node [] {1, 2, 3, 4}
Visiting node [1] {2, 3, 4}
Visiting node [1, 2] {3, 4}
Visiting node [1, 2, 3] {4}
New best [1, 2, 3] with value 6
Visiting node [1, 2, 3, 4] set()
Visiting node [1, 4] {2, 3}
New best [1, 4] with value 5
Visiting node [1, 4, 3] {2}
Visiting node [1, 4, 3, 2] set()
Visiting node [3] {1, 2, 4}
Visiting node [3, 2] {1, 4}
Visiting node [3, 2, 1] {4}
Visiting node [3, 2, 1, 4] set()
Visiting node [3, 4] {1, 2}
Visiting node [3, 4, 1] {2}
Visiting node [3, 4, 1, 2] set()
```

[5]: [[1, 4], 5]

As desired, the result is the sequence (1, 4) with lowest sum 5.

This version visits 15 nodes. Can we further prune the search space? (All together now: oh yes, we can!)

22.4.3 Avoid worse candidates

When using backtracking for an optimisation problem, we must think whether we can avoid generating candidates that won't lead to a better solution than the current one.

In this problem, as a sequence is extended, its sum grows because all extensions are positive numbers. When a sequence reaches or surpasses the sum of the current best solution, we know this sequence can't lead to a better solution, with a lower sum, so we can stop extending it.

To do this check, the `can_extend` function needs another parameter: the current best.

[6]: `def can_extend(item: int, candidate: list, best: list) -> bool:
 """Check if item can extend candidate to a better solution than
 →best."""
 return item % 2 != len(candidate) % 2 and item +_
 →value(candidate) < best[VALUE]`

Because of the extra parameter, we must change its call in `extend`.

[7]: `def extend(candidate: list, extensions: set, n: int, best: list) ->_
 →None:
 """Update best if a better extension of candidate is found."""`

(continues on next page)

(continued from previous page)

```

print("Visiting node", candidate, extensions)
if satisfies_range(candidate, n):
    candidate_value = value(candidate)
    if candidate_value < best[VALUE]:
        print("New best", candidate, "with value", candidate_
→value)
        best[SOLUTION] = candidate
        best[VALUE] = candidate_value
    for item in extensions:
        if can_extend(item, candidate, best): # changed line
            extend(candidate + [item], extensions - {item}, n, best)

best_sequence(4)

Visiting node [] {1, 2, 3, 4}
Visiting node [1] {2, 3, 4}
Visiting node [1, 2] {3, 4}
Visiting node [1, 2, 3] {4}
New best [1, 2, 3] with value 6
Visiting node [1, 4] {2, 3}
New best [1, 4] with value 5
Visiting node [3] {1, 2, 4}

[7]: [[1, 4], 5]

```

As you can see, only six of the previous 15 nodes are created and visited. Once a solution with sum 5 is found, partial candidates starting with 3 aren't generated because number 3 can only be followed by 2 or 4, both leading to sequences with sum at least 5 and thus not improving on the current best.

Is this the best (pun intended) we can do to prune the search space? (All together: oh no, it isn't!)

22.4.4 Start well

I mentioned earlier that we can start with any solution because the search continuously improves on it until it finds the best solution. However, now that the algorithm uses the best solution to not generate candidates leading to equally good or worse solutions, we should start with an initial solution close to a best one, to further prune the search space. The trick is to think of a good solution that is easy to construct.

Based on the fact that $(1, 4)$ is a best solution for $n = 4$, I choose $(1, n)$ or $(1, 2, n)$ as the initial solution, depending on whether n is even or odd. (Remember that $n > 2$.)

```
[8]: def best_sequence(n: int) -> list:
    """Return a lowest sum sequence that satisfies both constraints."""
    candidate = []

```

(continues on next page)

(continued from previous page)

```

extensions = set(range(1, n + 1))
if n % 2 == 0:
    solution = [1, n]
else:
    solution = [1, 2, n]
best = [solution, value(solution)]
extend(candidate, extensions, n, best)
return best

best_sequence(4)

Visiting node [] {1, 2, 3, 4}
Visiting node [1] {2, 3, 4}
Visiting node [1, 2] {3, 4}
Visiting node [3] {1, 2, 4}

[8]: [[1, 4], 5]

```

The final version only visits four of the initial version's 15 nodes: the search space has been reduced by over 70%!

To sum up, backtracking can solve optimisation problems by keeping track of the current best solution and its value, and by updating both when a better solution is found. For optimisation problems where extending a candidate worsens its value, e.g. makes the value larger but it's a minimisation problem, the search space can be pruned by stopping extending a candidate when its value is equal to or worse than the current best. Further pruning can be obtained by constructing an initial solution close to a best one.

22.5 Back to the TSP

In this section, we solve an optimisation problem with backtracking: the *travelling salesman problem* (TSP).

Function: `tsp`

Inputs: `graph`, a weighted undirected graph

Preconditions: `graph` is complete; the nodes are 0, 1, ...; the weights are positive

Output: `shortest`, a sequence of nodes of `graph`

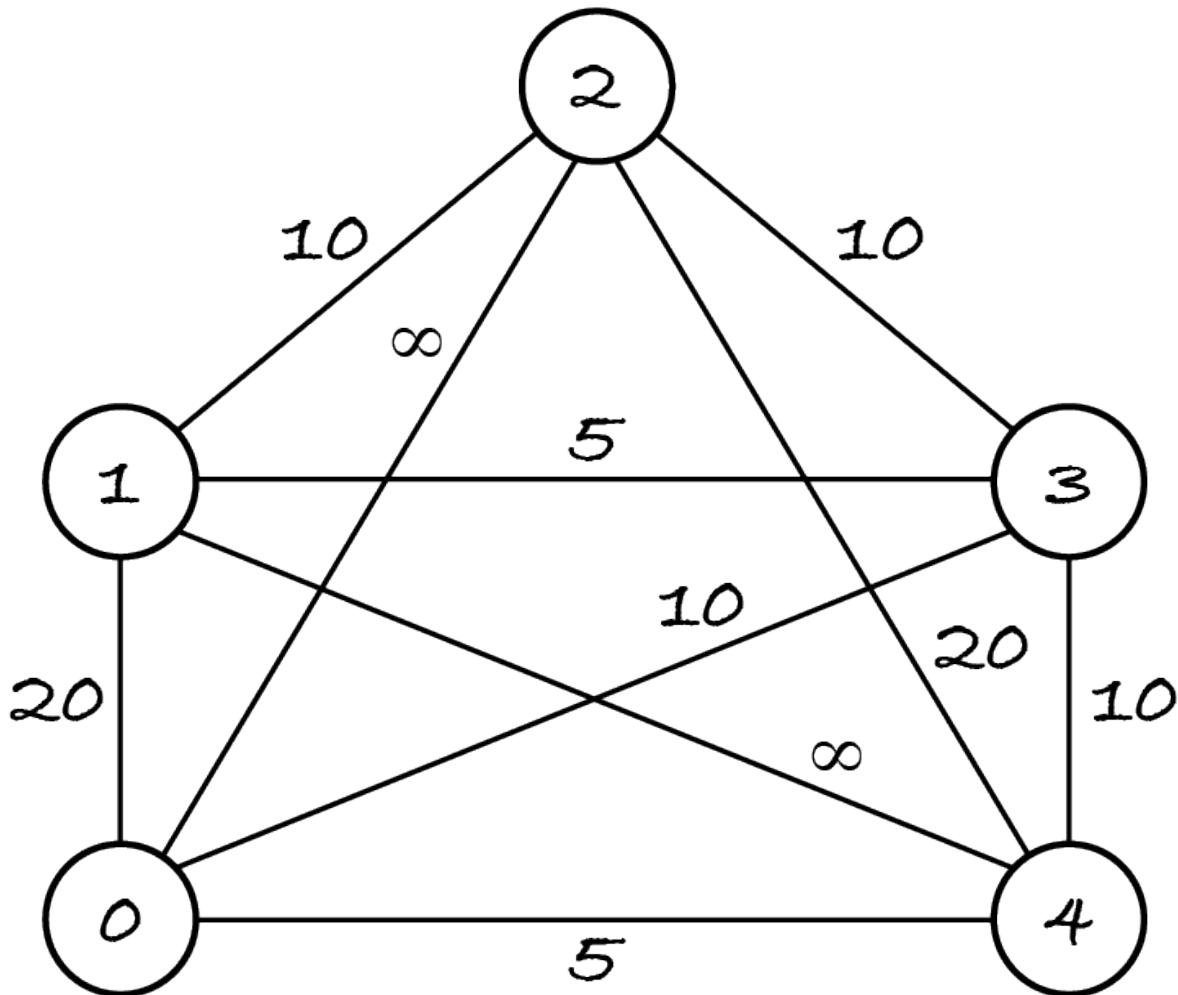
Postconditions:

- `shortest` is a tour that starts and ends at node 0
- no other tour of `graph` has a lower total weight than `shortest`

Real networks seldom have connections between all nodes, but we can model any network as a complete graph by adding the missing edges with infinite weight. This prevents them from being selected for a shortest tour.

Here's a graph to test the algorithm. Edges 0–2 and 1–4 have infinite weight, to show they're missing in the network being modelled.

Figure 22.5.1



The graph has $4! = 24$ tours that start and end at node 0, but only those of finite length exist in the network modelled by the graph: (0, 1, 2, 3, 4, 0), (0, 1, 2, 4, 3, 0), (0, 1, 3, 2, 4, 0), (0, 3, 1, 2, 4, 0) and the reverse sequences. The shortest tours are (0, 3, 1, 2, 4, 0) and (0, 4, 2, 1, 3, 0) with length 50.

```

[1]: import math

%run -i ../m269_digraph.py      # (un)weighted directed graph classes
%run -i ../m269_undirected.py   # undirected graph classes inherit
→from directed

example = WeightedUndirectedGraph()
for node in range(5):
    example.add_node(node)
example.add_edge(0, 1, 20)
example.add_edge(0, 2, math.inf)
    
```

(continues on next page)

(continued from previous page)

```
example.add_edge(0, 3, 10)
example.add_edge(0, 4, 5)
example.add_edge(1, 2, 10)
example.add_edge(1, 3, 5)
example.add_edge(1, 4, math.inf)
example.add_edge(2, 3, 10)
example.add_edge(2, 4, 20)
example.add_edge(3, 4, 10)
```

Previously, we started with high-level questions about what the items, the candidates and the solutions are, to have a good understanding of the problem before solving it. Another approach is to let the backtracking code template guide the questions.

22.5.1 The main function

The TSP is an optimisation problem on tours, which can be represented as sequences of nodes. So let's start with the main function template for optimisation problems on sequences.

```
SOLUTION = 0
VALUE = 1

def problem(instance: object) -> list:
    """Return the best solution and its value for the problem
    ↪instance."""
    candidate = []
    extensions = ...
    solution = ...
    best = [solution, value(solution)]
    extend(candidate, extensions, instance, best)
    return best
```

Let's go through it line by line.

The problem is the TSP and the instance is a weighted undirected graph, so the function header becomes:

```
def tsp(graph: WeightedUndirectedGraph) -> list:
```

Usually, the initial candidate is the empty sequence and the initial extensions form a set of items to be added to the sequence, each item occurring at most once in a solution. For the TSP problem that's not true: node 0 occurs at the start and end of each tour. To be able to generate tours, the root candidate must be sequence (0) and the extensions are all the nodes in the graph, so that node 0 can be added a second time to the sequence.

```
candidate = [0]
extensions = graph.nodes()
```

As for initialising the best solution, I could construct the tour $(0, 1, 2, \dots, n-1, 0)$ and compute

its length.

```
solution = sorted(graph.nodes()) + [0]
best = [solution, value(solution)] # function value to be written
```

However, I take the opportunity to show you a trick that works for any optimisation problem on sequences, especially when there's no easy way of constructing a good initial solution. We start with a 'pseudo-solution' (usually the empty sequence) and an infinitely high or low value, depending on whether it's a minimisation or maximisation problem. This guarantees that the first solution found is necessarily better.

The TSP is a minimisation problem, so we start with an infinitely high value.

```
solution = []
best = [solution, math.inf]
```

The problem only asks for the tour, not its length, so the last template line becomes:

```
return best[SOLUTION]
```

Let's put all this together and make the docstring and variable names less generic.

```
[2]: SOLUTION = 0
      VALUE = 1

def tsp(graph: WeightedUndirectedGraph) -> list:
    """Return a tour-length pair with shortest length.

    Preconditions:
    graph is complete, has nodes 0, 1, ..., and positive weights
    Postconditions: tour starts and ends at node 0
    """
    path = [0]
    nodes = graph.nodes()
    solution = []
    shortest = [solution, math.inf]
    extend(path, nodes, graph, shortest)
    return shortest[SOLUTION]
```

The backtracking algorithm, implemented by function `extend`, is mostly boilerplate. The problem-specific computations are in the auxiliary functions that compute the value of a candidate and check the constraints, so let's tackle them next.

22.5.2 The value function

For any optimisation problem, we must compute the value of a candidate to know whether it improves on the best solution so far. For the TSP, what is a candidate and what does the value function compute?

A candidate is a sequence of nodes representing a path starting at node 0. The function computes the total weight of the edges between consecutive nodes. The next exercise asks you to implement the value function. Before that, I recommend you uncomment and run the next code line to remind yourself of the available graph methods and their parameters.

```
[3]: # help(WeightedUndirectedGraph)
```

Exercise 22.5.1

For this and the following exercises, you're given a code template that you must adapt and complete for the TSP. Adapting the code means to:

- replace the generic docstrings and identifiers with problem-specific ones (you can press F, not Shift-F, in command mode to find and replace text in the current cell or in all cells)
- remove any unnecessary parameters.

Adapt and complete the following value function template for the TSP.

```
[4]: def value(candidate: list, instance: object) -> int:  
    """Return the value of the candidate sequence for the problem  
    ↪instance."""  
    pass # replace with your code
```

```
value([0, 1, 2, 3, 4, 0], example) == 55
```

Hint Answer

22.5.3 Checking the constraints

Two auxiliary functions check the global and local constraints on candidates. For the TSP, candidates are paths (sequences of nodes) starting at node 0, and solutions are candidates representing tours. The questions to think about are as follows.

- Can partial candidates be solutions?
-

A tour has all the graph's nodes, so solutions must be complete candidates; that is, when there's no further node to add to the sequence.

- What are the constraints on a complete candidate for it to be a solution?
-

The candidate path ends with node 0 and each node has an edge to the next one.

- Which constraints are local (can be checked on partial candidates) and which are global (must be checked on complete candidates)?
-

The existence of edges between consecutive nodes is a local constraint. The sequence ending with node 0 is a global constraint.

Exercise 22.5.2

Adapt and complete the next code template for the TSP.

```
[5]: def satisfies_global(candidate: list, instance: object) -> bool:  
    """Check if the candidate satisfies the global constraints."""  
    pass # replace with your code
```

Answer

Section 22.4.3 mentions that if extending a candidate worsens its value, then it shouldn't be further extended when it reaches the best value so far.

For the TSP, does extending a candidate worsen its value?

The value of a candidate path is its length: the total weights of its edges. The problem definition tells us weights are positive, so extending a path increases its length, which for a minimisation problem is a worse value.

Exercise 22.5.3

Adapt and complete the next code template for the TSP.

```
[6]: def can_extend(item: object, candidate: list, instance: object, best:  
    -> list) -> bool:  
    """Check if item can extend candidate into a better solution  
    than best."""  
    # replace ... with a check for the local constraints  
    # use < for a minimisation problem  
    return ... and value(candidate + [item]) > best[VALUE]
```

Hint Answer

22.5.4 The backtracking function

Now we have all auxiliary functions in place for the backtracking algorithm.

Exercise 22.5.4

Adapt the next code template to the TSP. Don't forget to change the calls to `satisfies_global`, `value` and `can_extend` if you changed their names in the cells above.

```
[7]: def extend(candidate: list, extensions: set, instance: object, best:  
    -> list) -> None:  
    """Update best if candidate is a better solution, otherwise  
    extend it."""  
    print("Visiting node", candidate, extensions)  
    # remove the next line if partial candidates can be solutions  
    if len(extensions) == 0:
```

(continues on next page)

(continued from previous page)

```

if satisfies_global(candidate, instance):
    candidate_value = value(candidate, instance)
    # in the next line, use < for minimisation problems
    if candidate_value > best[VALUE]:
        print("New best with value", candidate_value)
        best[SOLUTION] = candidate
        best[VALUE] = candidate_value
    for item in extensions:
        if can_extend(item, candidate, instance, best):
            extend(candidate + [item], extensions - {item}, instance,
→ best)
    
```

Answer

Finally uncomment the next line and run it to check your solution.

```
[8]: # tsp(example) in [ [0, 3, 1, 2, 4, 0], [0, 4, 2, 1, 3, 0] ]
```

The current algorithm still does much wasted work. It keeps extending paths where the second node 0 appears early on, only to later fail the global constraint test. The next exercise asks you to avoid generating such paths.



Note: After applying backtracking, look at the candidates generated and think if there are problem-specific ways of further pruning the search space.

Exercise 22.5.5

Make a copy of your `extend` function and change it so that a path is extended with node 0 only if it's the last remaining extension. This guarantees that all complete candidates end with node 0 and you can remove the call to the global constraint check.

Run the example again to see a substantial reduction in the search space.

Answer

22.6 Generate subsets

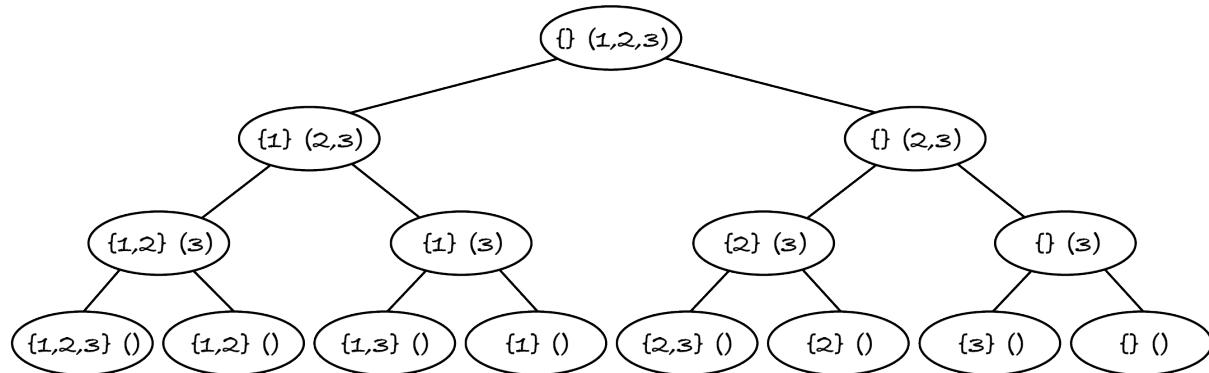
Backtracking can also solve problems on sets instead of sequences. The only thing that changes is the core tree-traversal algorithm. The rest – handling constraints, computing the value of each candidate, keeping track of the best solution so far, etc. – remains the same. So let's see how a tree traversal can generate all subsets of the given items.

When we tried to *greedily solve the knapsack problem*, we first put all items into a sequence, sorted from best to worst. The algorithm then picked one item at a time. If the knapsack still had capacity for that item, it was added to the output set; otherwise it was skipped.

A tree-traversal algorithm to generate all subsets works in the same way. It starts with an empty candidate set and with all items in a sequence of extensions. Recursively, the algorithm takes each item in turn from the extensions and adds it to the candidate set or skips it.

Here's the tree of candidate–extension pairs for generating subsets of $\{1, 2, 3\}$. In each node, the candidate is the subset constructed so far and the sequence has the numbers yet to consider.

Figure 22.6.1



The root's candidate is the empty set and the root's extensions are all items (here the numbers from 1 to 3), in any order. Each node has two children, both with the first extension removed. The left child adds the extension to the candidate; the right child doesn't.

Like for the tree of permutations, the complete candidates, with empty extensions, are in the leaves of this tree. The other nodes have partial candidates. Due to skipping items, some subsets occur multiple times, e.g. two nodes have subset $\{1, 2\}$ and four nodes have the empty set. We only consider the leaves because only then do we know that no further items will be added.

The tree traversal for subsets looks like this, using $s[i:]$ as shorthand for $s[i:len(s)]$ ([Section 4.9.1](#)).

```
[1]: def extend(candidate: set, extensions: list, solutions: list) -> None:
    """Extend candidate with all subsets of extensions and add to solutions."""
    print("Visiting node", candidate, extensions)
    if len(extensions) == 0:  # complete candidate
        solutions.append(candidate)
    else:
        item = extensions[0]  # head of sequence
        rest = extensions[1:]  # tail of sequence
        extend(candidate.union({item}), rest, solutions)  # add item
        extend(candidate, rest, solutions)  # skip item
```

Like for generating permutations, the base case is when the extensions are empty and the reduction step (the assignment to `rest`) removes one extension.

Like for generating permutations, we must start the algorithm in the root node.

```
[2]: def all_subsets(n: int) -> list:
    """Return all subsets of 1, ..., n in the order generated."""
    candidate = set()
    extensions = list(range(1, n + 1))
    solutions = []
    extend(candidate, extensions, solutions)
    return solutions

print("Subsets:", all_subsets(3))

Visiting node set() [1, 2, 3]
Visiting node {1} [2, 3]
Visiting node {1, 2} [3]
Visiting node {1, 2, 3} []
Visiting node {1, 2} []
Visiting node {1} [3]
Visiting node {1, 3} []
Visiting node {1} []
Visiting node set() [2, 3]
Visiting node {2} [3]
Visiting node {2, 3} []
Visiting node {2} []
Visiting node set() [3]
Visiting node {3} []
Visiting node set() []
Subsets: [{1, 2, 3}, {1, 2}, {1, 3}, {1}, {2, 3}, {2}, {3}, set()]
```

If you follow how nodes are visited in the tree diagram, you'll see the algorithm is making a pre-order traversal. It chooses to add the item (left subtree) before choosing to skip it (right subtree), so the full subset $\{1, 2, 3\}$ is generated first and the empty subset is generated last.

The algorithm could of course first choose to skip and then choose to add the item. It would generate the same subsets, in reverse order. You can swap the order of the code lines that add and skip an item, to see for yourself.

Changing the order of the extensions, e.g. `extensions = list(range(n, 0, -1))`, also changes the order in which subsets are generated. I'll leave it to you to try it out.

Now that we have a recursive algorithm to generate each subset incrementally, we can search for a best subset that satisfies some constraints in exactly the same way as we did for sequences. The next section provides an example: the knapsack problem.

22.7 Back to the knapsack

To recap, the 0/1 knapsack problem is as follows.

Function: 0/1 knapsack

Inputs: *items*, a set of pairs of integers; *capacity*, an integer

Preconditions: no integer is negative; each pair represents a weight and a value

Output: *packed*, a set of pairs of integers

Postconditions:

- *packed* is a subset of *items*
- the sum of the weights in *packed* doesn't exceed *capacity*
- the sum of the values in *packed* is largest among all sets satisfying the previous two conditions

I'll use the following example:

```
[1]: ITEMS = { (1, 2), (2, 3), (3, 4), (4, 20), (5, 30) }
```

If the knapsack has a capacity of 4, it can hold the first and second items (weight $1 + 2 = 3$, value $2 + 3 = 5$) or the first and third items (weight $1 + 3 = 4$, value $2 + 4 = 6$) or the fourth item (weight 4, value 20). The latter is the desired output, because it has the largest value.

Let's follow the same procedure as before to solve this problem, with 'stop and think' lines for you to think along.

22.7.1 The problem

The knapsack problem is obviously an optimisation problem on subsets of items, where the value of the items in the subset is the quantity being maximised. Each candidate is the subset of items already put in the knapsack and the corresponding extensions are the items yet to consider.

1. What are the global and local constraints for this problem?
 2. Can partial candidates be solutions or only complete candidates?
-
1. The only constraint is that the weight of the items in the subset cannot exceed the knapsack's capacity. It's a local constraint because it can be checked as each item to be added is considered.
 2. As mentioned in the previous section, although partial candidates are subsets too, the solutions are the complete candidates, after all items have been considered.

We can start implementing the solution, from the auxiliary functions towards the backtracking and main functions.

22.7.2 The value function

Every optimisation problem needs a function to compute the value of a solution, so let's start with that. As usual, I define constants for the indices of pairs, to make the code more readable.

```
[2]: WEIGHT = 0  
VALUE = 1
```

(continues on next page)

(continued from previous page)

```
def value(items: set) -> int:
    """Return the total value of the items."""
    total = 0
    for item in items:
        total = total + item[VALUE]
    return total
```

22.7.3 The constraints functions

Next are the auxiliary functions to check the global and local constraints. Looking at the answers to the earlier questions, how does this problem differ from the other problems in this chapter?

There are no global constraints. Any candidate that satisfies the local constraint (fits the capacity) is a solution, but perhaps not a best one.

This means we don't need a `satisfies_global` function. We must only check if an item can extend a candidate (the items already in the knapsack) towards a better solution than the current one. Let's break that down into two parts.

How can we check whether an item can extend a candidate towards a solution? Or put differently, how do we know if there's no point in extending the given candidate with the given item?

If the weight of the item plus the weight of the candidate exceeds the capacity, then the item shouldn't be added to the knapsack.

Now let's assume that an item can extend a candidate towards a solution. Is there a way to know it won't lead to a better solution than the current best? (Hint: does extending a candidate worsen its value?)

The problem definition above states that items cannot have negative values. If an item *can* extend a candidate, we *must* extend it, because the new item may lead to a better (i.e. higher value) solution. There is no way of pruning the search space early.

In summary, an item can extend a candidate if the sum of their weights doesn't exceed the capacity.

[3]:

```
def can_extend(item: tuple, candidate: set, capacity: int) -> bool:
    """Check if adding the item to candidate won't exceed the
    capacity."""
    total = item[WEIGHT]
    for another_item in candidate:
        total = total + another_item[WEIGHT]
    return total <= capacity
```

22.7.4 The backtracking function

Here's the template for the backtracking function for optimisation problems on sets. The `instance` variable stands for the inputs of the problem, which are passed on to the auxiliary functions.

```
SOLUTION = 0
VALUE = 1

def extend(candidate: set, extensions: list, instance: object, best:_
→list) -> None:
    """Update best if candidate is a better solution, then try to_
→extend it."""
    print('Visiting node', candidate, extensions)
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            candidate_value = value(candidate, instance)
            if candidate_value == best[VALUE]: # replace == with <_
→or >
            print('New best with value', candidate_value)
            best[SOLUTION] = candidate
            best[VALUE] = candidate_value
    else:
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, instance, best):
            extend(candidate.union({item}), rest, instance, best)
            extend(candidate, rest, instance, best)
```

Let's think what changes are needed for the knapsack problem.

Which comparison should be used: less than or greater than?

It's a maximisation problem so the candidate is the new best if its value is greater than the current one.

Looking at the auxiliary functions written before, can any code or parameters be removed?

Yes, the `value` function only needs the `candidate` parameter, the `can_extend` function only needs the `item`, `candidate` and `capacity`, and the `satisfies_global` function isn't needed at all.

Finally, given the previous changes, what should the `instance: object` parameter in the function header be replaced with?

It should be `capacity: int`, which is needed by `can_extend`.

Here's the backtracking function for the knapsack problem.

```
[4]: SOLUTION = 0
VALUE = 1

def extend(candidate: set, extensions: list, capacity: int, best:_
→list) -> None:
    """Update best if candidate is a better solution, then try to_
→extend it."""
    print("Visiting node", candidate, extensions)
    if len(extensions) == 0:
        candidate_value = value(candidate)
        if candidate_value > best[VALUE]:
            print("New best with value", candidate_value)
            best[SOLUTION] = candidate
            best[VALUE] = candidate_value
    else:
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, capacity):
            extend(candidate.union({item}), rest, capacity, best)
extend(candidate, rest, capacity, best)
```

22.7.5 The main function

We finally have to think of what the main function needs to do.

What are the candidate and the extensions of the root node?

The initial candidate is the empty set and the extensions are all the items given in the input, in a sequence.

What is a possible initial best solution?

We need a solution that can be easily constructed. The only one I could think of is the empty set: it's a solution of every problem instance though hardly ever the best one (unless no item fits in the knapsack).

The main function is thus:

```
[5]: def knapsack(items: set, capacity: int) -> list:
    """Return a subset of items and their total value.

    Preconditions:
    - items is a set of weight-value pairs, both integers
    - no integer is negative
    Postconditions:
    """
```

(continues on next page)

(continued from previous page)

```

- the output is a set-integer pair
- total weight of the output items <= capacity
- no other subset of items has higher value and fits the capacity
"""

candidate = set()
extensions = list(items)
solution = set()
best = [solution, value(solution)]
extend(candidate, extensions, capacity, best)
return best

```

Let's solve the example at the start of the notebook.

```
[6]: knapsack(ITEMS, 4)

Visiting node set() [(2, 3), (1, 2), (3, 4), (5, 30), (4, 20)]
Visiting node {(2, 3)} [(1, 2), (3, 4), (5, 30), (4, 20)]
Visiting node {(2, 3), (1, 2)} [(3, 4), (5, 30), (4, 20)]
Visiting node {(2, 3), (1, 2)} [(5, 30), (4, 20)]
Visiting node {(2, 3), (1, 2)} [(4, 20)]
Visiting node {(2, 3), (1, 2)} []
New best with value 5
Visiting node {(2, 3)} [(3, 4), (5, 30), (4, 20)]
Visiting node {(2, 3)} [(5, 30), (4, 20)]
Visiting node {(2, 3)} [(4, 20)]
Visiting node {(2, 3)} []
Visiting node set() [(1, 2), (3, 4), (5, 30), (4, 20)]
Visiting node {(1, 2)} [(3, 4), (5, 30), (4, 20)]
Visiting node {(1, 2), (3, 4)} [(5, 30), (4, 20)]
Visiting node {(1, 2), (3, 4)} [(4, 20)]
Visiting node {(1, 2), (3, 4)} []
New best with value 6
Visiting node {(1, 2)} [(5, 30), (4, 20)]
Visiting node {(1, 2)} [(4, 20)]
Visiting node {(1, 2)} []
Visiting node set() [(3, 4), (5, 30), (4, 20)]
Visiting node {(3, 4)} [(5, 30), (4, 20)]
Visiting node {(3, 4)} [(4, 20)]
Visiting node {(3, 4)} []
Visiting node set() [(5, 30), (4, 20)]
Visiting node set() [(4, 20)]
Visiting node {(4, 20)} []
New best with value 20
Visiting node set() []

[6]: [{(4, 20)}, 20]
```

Compared to an exhaustive search that generates all $2^5 = 32$ subsets of the five items and tests

each subset for whether it fits the knapsack and has a better value, the backtracking approach only generates seven subsets (those with empty extension sequences in the printout). However, many partial candidates are generated. Fortunately, there's a way to further prune the search space.

22.7.6 Sort extensions

When searching a store for products below £20, we *sorted the products* by ascending price. That allowed us to stop the linear search as soon as we found a product costing £20 or more. We can apply the same idea here.

Let's sort the items by ascending weight. If adding the current item to a candidate exceeds the capacity, so will adding any subsequent item in the extensions sequence, because they weigh even more.

At the moment, if adding an item exceeds the capacity, we skip only *that* item. Sorting the items by ascending weight allows us to skip *all* the remaining extensions too: a massive reduction in the search space.

Here's the new main function. Items are weight–value pairs so Python's lexicographic sorting of tuples puts them in ascending weight. I don't repeat the docstring.

```
[7]: def knapsack(items: set, capacity: int) -> list: # noqa: D103
    candidate = set()
    extensions = sorted(items) # changed line
    solution = set()
    best = [solution, value(solution)]
    extend(candidate, extensions, capacity, best)
    return best
```

If a partial candidate can't be extended because all remaining extensions go over the capacity, the candidate may still be the best solution so far. This means we have to check partial candidates against the best solution, not just complete candidates. Here's the new backtracking function.

```
[8]: def extend(candidate: set, extensions: list, capacity: int, best: list) -> None:
    """Update best if candidate is a better solution, then try to extend it."""
    print("Visiting node", candidate, extensions)
    candidate_value = value(candidate)
    if candidate_value > best[VALUE]:
        print("New best with value", candidate_value)
        best[SOLUTION] = candidate
        best[VALUE] = candidate_value
    if len(extensions) > 0: # changed line
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, capacity):
            extend(candidate.union({item}), rest, capacity, best)
            extend(candidate, rest, capacity, best) # changed line
```

Notice the changes to the previous version.

- I remove the check at the beginning for a complete candidate (no extensions), because now partial candidates can be solutions.
- I check if there are any extensions before I look at the next item.
- I indent the last line of code, the one which skips the item.

The last change is subtle but profound. It implements the skipping of all further extensions if the current one can't extend the candidate. Let's see the impact on the search space.

```
[9]: knapsack(ITEMS, 4)

Visiting node set() [(1, 2), (2, 3), (3, 4), (4, 20), (5, 30)]
Visiting node {(1, 2)} [(2, 3), (3, 4), (4, 20), (5, 30)]
New best with value 2
Visiting node {(2, 3), (1, 2)} [(3, 4), (4, 20), (5, 30)]
New best with value 5
Visiting node {(1, 2)} [(3, 4), (4, 20), (5, 30)]
Visiting node {(1, 2), (3, 4)} [(4, 20), (5, 30)]
New best with value 6
Visiting node {(1, 2)} [(4, 20), (5, 30)]
Visiting node set() [(2, 3), (3, 4), (4, 20), (5, 30)]
Visiting node {(2, 3)} [(3, 4), (4, 20), (5, 30)]
Visiting node set() [(3, 4), (4, 20), (5, 30)]
Visiting node {(3, 4)} [(4, 20), (5, 30)]
Visiting node set() [(4, 20), (5, 30)]
Visiting node {(4, 20)} [(5, 30)]
New best with value 20
Visiting node set() [(5, 30)]
```

[9]: [{(4, 20)}, 20]

The search space has almost halved: only 13 of the previous 24 nodes are created and visited. For example, partial candidate $\{(1, 2), (2, 3)\}$ is not extended because any further item exceeds the capacity.



Note: If possible, order the extensions sequence so that if one item in the sequence can't extend a candidate, none of the following can.

Exercise 22.7.1

The original problem asks for *any* subset of the items that maximises the value and fits the knapsack. Imagine we add one postcondition: the returned subset should be as small as possible, i.e. we want to pack the fewest items that maximise the value and fit in the knapsack.

For example, let the items be $\{(1, 2), (2, 3), (4, 5)\}$ and the capacity be 4. The largest possible value is 5 and can be obtained in two ways: pack the two items $\{(1, 2), (2, 3)\}$ or pack the single

item $\{(4, 5)\}$. Any of these two subsets is a solution to the original problem but only the latter subset is a solution for the new problem, as it has fewer items.

What changes would be required to the `extend` function?

Hint Answer

22.8 Summary

Backtracking is a search technique that is usually much more efficient than exhaustive search because it only generates a fraction of all candidates.

In M269, we apply backtracking only to constraint satisfaction and optimisation problems on sequences of non-duplicate items or subsets of items. This includes some problems on strings and on paths in a grid or graph.

Backtracking starts with an empty candidate sequence or set and extends it one item at a time. In M269, we only solve problems where the possible extensions are known in advance. If extending a candidate with a particular item cannot lead to a solution, or a better solution, then the algorithm goes back and tries a different item or a different candidate.

At the core of backtracking is a recursive pre-order traversal of the tree of all candidates and their yet unexplored extensions. From the root to the leaves, items are taken from the extensions and added to the candidate. The leaves have the **complete candidates**: they can't be further extended because there are no extensions left. The other nodes have the **partial candidates**, which can still be extended. There are two core traversals, for generating permutations and subsets.

Besides the traversal, a backtracking algorithm does the following:

1. Check if it's worth extending the candidate with a chosen item. This check is the key ingredient to prune the search space and to turn an exhaustive traversal into a backtracking algorithm.
2. Check if a candidate is a solution.
3. For optimisation problems, compute the value of each solution found and update the current best solution as better ones are found.

The first check covers the **local constraints** of the problem, and the second check the **global constraints**. The first check prunes the search space if extending the candidate breaks the local constraints or worsens the candidate's value: it gets larger in a minimisation problem or smaller in a maximisation problem.

For some problems on subsets, the search space can be further pruned by sorting in advance the items so that when one can't extend a candidate, neither can any of the subsequent items.

Backtracking can be applied in a systematic way. First think about what kind of problem it is, what the items, candidates and extensions represent, what the global and local constraints are, and how to prune the search space. Then modify the applicable code template listed in [Section 25.2](#).

CHAPTER 23

DYNAMIC PROGRAMMING

We have solved various problems in a recursive way. In many, the solution to a problem was obtained by solving *one* subproblem (i.e. a smaller instance of the same problem) and so the algorithm made a single recursive call. For example, if the problem is to obtain the *length of a list*, we can solve it recursively by solving the subproblem of computing the length of its tail. Another example is *binary search*: to solve the problem of finding an item in a sorted sequence, it solves the subproblem of finding the item in either the left or right half (it never checks both halves).

Other problems were divided into *multiple independent* subproblems, and we applied a divide-and-conquer algorithm that made as many recursive calls as there were subproblems (usually two). For example, *quicksort* and *merge sort* divide the input sequence into two separate parts and sort each one, before combining the sorted parts into a sorted sequence. Algorithms on *binary trees* are another example: they process the left and right subtrees separately and then combine both results with the root's item.

However, some problems are naturally split into *overlapping* subproblems, where two or more subproblems have a common subproblem. While an algorithm with multiple recursion can elegantly solve such problems, it's very inefficient as it solves common subproblems repeatedly.

Dynamic programming is a technique that makes a space–time tradeoff to solve such problems more efficiently. It solves each subproblem only once and stores its solution in a subproblem–solution map. When a common subproblem occurs again, its solution is looked up instead of being recomputed. This chapter covers some classic dynamic programming problems to illustrate the technique. (The term ‘dynamic programming’ is unrelated to dynamic arrays.)

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter introduces the dynamic programming technique.
- Analyse the complexity of algorithms to support software design choices – you will learn how to analyse the complexity of dynamic programming.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

23.1 Fibonacci

The infinite integer sequence 1, 1, 2, 3, 5, 8, 13, 21, ... is known as the Fibonacci sequence. The first two numbers are 1 and thereafter each number is the sum of the previous two. (You can search online for the history and relevance of this sequence, if you're interested.)

This section uses the problem of computing the n -th Fibonacci number, for $n \geq 1$, to explain the need for dynamic programming and how it works. We'll use these tests:

```
[1]: from algoesup import check_tests, test

fibonacci_tests = [
    # case,      n,   n-th Fibonacci number
    ('n = 1',    1,    1),
    ('n = 2',    2,    1),
    ('n = 6',    6,    8),
    ('n = 10',   10,   55)
]

check_tests(fibonacci_tests, [int, int])
OK: the test table passed the automatic checks.
```

23.1.1 Recursive

To develop a dynamic programming algorithm we must start with a recursive definition of the problem at hand, which is:

$$\begin{aligned} \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(2) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \text{ for } n > 2 \end{aligned}$$

The recursive algorithm follows the definition directly.

```
[2]: def fibonacci_rec(n: int) -> int:
    """Return the n-th Fibonacci number, computed recursively.

    Preconditions: n > 0
    """
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci_rec(n - 1) + fibonacci_rec(n - 2)

test(fibonacci_rec, fibonacci_tests)
Testing fibonacci_rec...
Tests finished: 4 passed (100%), 0 failed.
```

Unfortunately this algorithm quickly gets very slow.

```
[3]: for n in range(20, 25):
    %timeit -r 3 fibonacci_rec(n)

480 µs ± 142 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
˓→each)
776 µs ± 108 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
˓→each)
1.26 ms ± 646 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops_
˓→each)
2.03 ms ± 1.17 µs per loop (mean ± std. dev. of 3 runs, 100 loops_
˓→each)
3.29 ms ± 1.65 µs per loop (mean ± std. dev. of 3 runs, 100 loops_
˓→each)
```

Computing the twentieth Fibonacci number should only require a few additions, done in nanoseconds, and yet it's taking hundreds of microseconds! The run-time is almost doubling every time n increases. This indicates exponential complexity in the value of n .

We can confirm this with a *recursive definition of the run-time*. Computing the first two Fibonacci numbers takes constant time. Every other value takes the time to compute the previous two values and some further constant time for adding them together.

$$T(1) = \Theta(1)$$

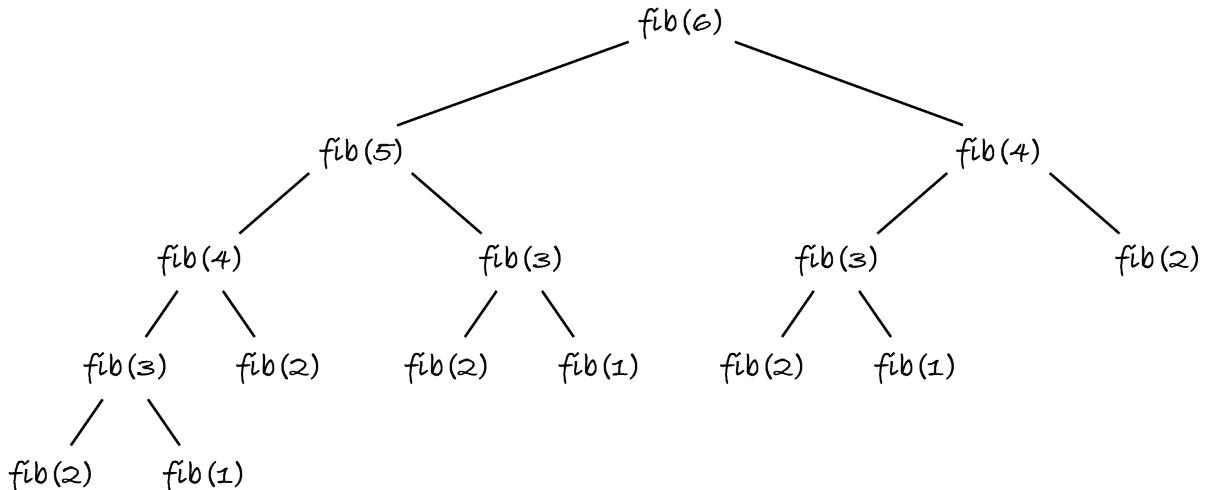
$$T(2) = \Theta(1)$$

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \text{ for } n > 2$$

Computing $\text{fibonacci}(n-1)$ involves computing $\text{fibonacci}(n-2)$, so the former takes longer to compute than the latter: $T(n-1) > T(n-2)$. This means that $T(n) > 2 \times T(n-2)$: the run-time for n is more than double that for $n-2$, as the above run-times confirm: the 24th number takes more than double the time to compute the 22nd, which in turn takes more than double the time for the 20th. If a fixed increment of the input (here, by two) more than doubles the run-time, this means the complexity is exponential.

The exponential run-time is due to the overlapping subproblems. (A subproblem is a smaller instance of the same problem.) A diagram shows this best. Here are the recursive calls for $n = 6$, abbreviating the function name to 'fib' to avoid cluttering the diagram.

Figure 23.1.1



The nodes for $n = 3$ and $n = 4$ have a common child: $n = 2$. Hence the two problem instances overlap: they require solving a common smaller instance. More generally, problem instances n and $n-1$ overlap because they have common subproblem $n-2$. The diagram shows that the algorithm is repeatedly solving the common subproblems. For example, the base case $n = 2$ is computed five times!

23.1.2 Top-down

A simple way to reduce the run-time is to store the result for each subproblem in a **cache**. When the solution of a subproblem is computed, it's put in the cache so that subsequent calls for the same subproblem simply look up the solution instead of recomputing it.

The cache is a data structure that maps subproblems to solutions and so any implementation of the *map ADT* will do, e.g. a hash table where the keys are the subproblems and the values are the corresponding solutions. Once a subproblem–solution pair is added to the cache, it's never removed or modified. So the cache only has to support two ADT operations: membership (is a subproblem already in the cache?) and lookup (what's the solution for this subproblem?).

Adding a cache to the recursive algorithm can be done in a systematic way. Here's the recursive function again, without repeating the docstring:

```
def fibonacci(n: int) -> int:  
    if n == 1 or n == 2:  
        return 1  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Now the version with a cache. First we check if the subproblem is in the cache. If it isn't, we follow the recursive algorithm and store the computed solution in the cache. Finally we return the cached solution, which was already there or has just been computed.

```
def fibonacci(n: int, cache: dict) -> int:
    if n not in cache:
        if n == 1 or n == 2:
            cache[n] = 1
```

(continues on next page)

(continued from previous page)

```

else:
    cache[n] = fibonacci(n - 1, cache) + fibonacci(n - 2, ↵
→cache)
return cache[n]
    
```

This function has a second parameter, so we can't reuse the tests written above. Moreover, it allows the caller to initially pass any dictionary, which could lead to wrong results.

A better version hides the cache from the user: the main function creates an empty cache and an inner recursive function uses it. (Remember that inner functions can access the variables of outer functions.)

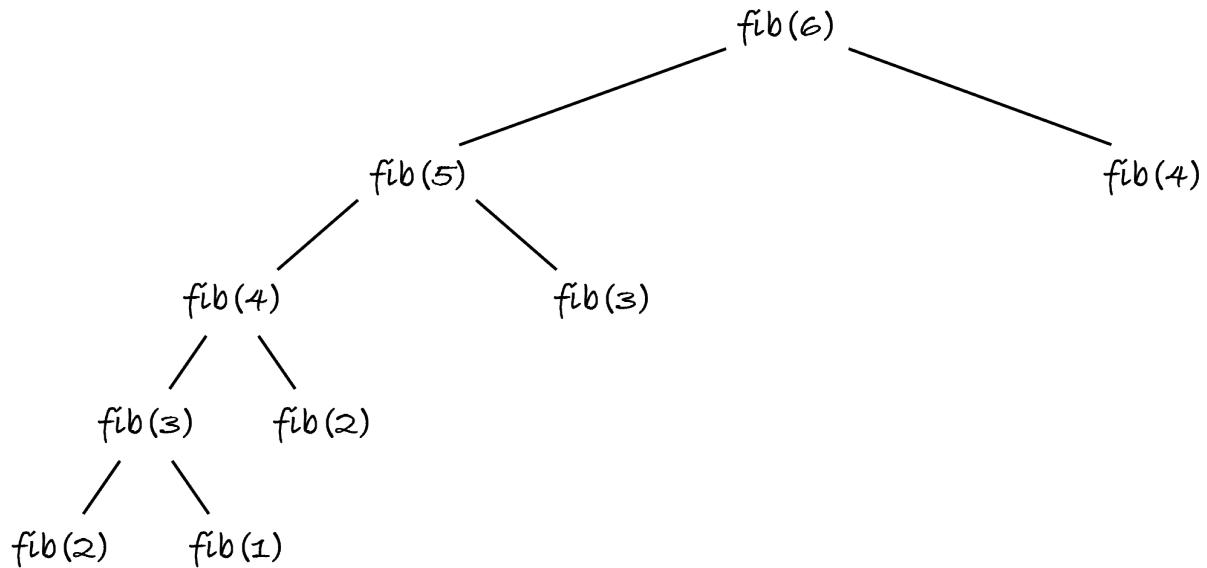
```
[4]: def fibonacci_td(n: int) -> int: # noqa: D103
    def fib(n: int) -> int:
        """Compute and cache the value if necessary. Return the
→cached value."""
        if n not in cache:
            if n == 1 or n == 2:
                cache[n] = 1
            else:
                cache[n] = fib(n - 1) + fib(n - 2)
        return cache[n]

    cache = dict()
    return fib(n)

test(fibonacci_td, fibonacci_tests)
Testing fibonacci_td...
Tests finished: 4 passed (100%), 0 failed.
```

In the expression `fib(n-1) + fib(n-2)`, the call `fib(n-2)` takes constant time because it just retrieves the value from the cache: the call `fib(n-1)` already computed `fib(n-2)`. So the second call to `fib(n-2)` makes no further recursive calls. The call tree of the inner function for $n = 6$ becomes:

Figure 23.1.2



The recursive complexity definition becomes

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

$$T(n) = T(n-1) + \Theta(1)$$

As explained in [Section 13.1.1](#), this results in $T(n) = \Theta(n)$. Caching the subproblems' solutions lowered the complexity from exponential to linear! Doubling the value of n confirms that the run-times more or less double too.

```
[5]: for n in (8, 16, 32, 64):
    %timeit -r 3 fibonacci_td(n)

979 ns ± 0.04 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
1.97 µs ± 2.63 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
3.95 µs ± 9.36 ns per loop (mean ± std. dev. of 3 runs, 100,000 loops each)
7.97 µs ± 6.49 ns per loop (mean ± std. dev. of 3 runs, 100,000 loops each)
```

The technique of caching the solutions as they're recursively computed is known as **top-down dynamic programming**. It starts with the given input (represented by the root node of the call tree) and then recursively solves smaller instances until it reaches a leaf of the call tree, which is either a base case or an already solved (i.e. cached) subproblem.



Info: Top-down dynamic programming is also called recursion with memoisation (not memorisation).

23.1.3 Bottom-up

Let's look again at the full call tree in Figure 23.1.1: it shows for each problem node the two children subproblems it depends on. The leaves are the base cases ($n = 1$ and $n = 2$), which depend on nothing.

We can solve the problem instances bottom-up: knowing the solutions for the base cases allows us to compute the solution for their parent ($n = 3$), which in turn allows us to compute the solution for its parent ($n = 4$) and so on until we reach the root.

This can be done iteratively, by adding to the cache the solutions to $n = 1, 2, 3, \dots$, in this order. This ensures that when computing the n -th Fibonacci number, the two previous numbers are already in the cache.

```
[6]: def fibonacci_bu(n: int) -> int: # noqa: D103
    # create cache with base cases fibonacci(1) = fibonacci(2) = 1
    cache = {1: 1, 2: 1}
    for number in range(3, n + 1):
        cache[number] = cache[number - 1] + cache[number - 2]
    return cache[n]

test(fibonacci_bu, fibonacci_tests)

Testing fibonacci_bu...
Tests finished: 4 passed (100%), 0 failed.
```

The complexity is linear because the loop executes $n-2$ additions. Even though the complexity is the same as for top-down dynamic programming, this version runs faster, because there are no recursive calls and no checks for cached solutions (since they are computed in the order needed).

```
[7]: for n in (8, 16, 32, 64):
    %timeit -r 3 fibonacci_bu(n)

447 ns ± 0.196 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
970 ns ± 0.919 ns per loop (mean ± std. dev. of 3 runs, 1,000,000 loops each)
2.03 µs ± 5.34 ns per loop (mean ± std. dev. of 3 runs, 100,000 loops each)
4.09 µs ± 5.03 ns per loop (mean ± std. dev. of 3 runs, 100,000 loops each)
```

Iteratively filling the cache from the smaller to the larger problem instances, so that each one is solved using already cached solutions, is known as **bottom-up dynamic programming**.

23.1.4 With arrays

To calculate the n -th Fibonacci number, we must solve the problem instances $1, 2, 3, \dots, n$. Since these are natural numbers, we can use them to index an array of solutions, instead of hashing them to retrieve solutions from a dictionary.

The array must be initialised with impossible solutions, so that it's easy to check whether a solution has been cached or not. Fibonacci numbers are always positive, so the cache can be initialised with zeros or with negative numbers. I choose zeros.

I only have to change two lines of code to replace the dictionary in the top-down approach with a Python list, used as a static array.

```
[8]: def fibonacci_tda(n: int) -> int: # noqa: D103
    def fib(n: int) -> int:
        if cache[n] == 0: # changed
            if n == 1 or n == 2:
                cache[n] = 1
            else:
                cache[n] = fib(n - 1) + fib(n - 2)
        return cache[n]

    cache = [0] * (n + 1) # changed
    return fib(n)

test(fibonacci_tda, fibonacci_tests)
Testing fibonacci_tda...
Tests finished: 4 passed (100%), 0 failed.
```

And now the bottom-up approach. I only change the cache creation.

```
[9]: def fibonacci_bua(n: int) -> int: # noqa: D103
    cache = [0] * (n + 1)
    cache[1] = 1 # first base case
    if n > 1:
        cache[2] = 1 # second base case
    for number in range(3, n + 1):
        cache[number] = cache[number - 1] + cache[number - 2]
    return cache[n]

test(fibonacci_bua, fibonacci_tests)
Testing fibonacci_bua...
Tests finished: 4 passed (100%), 0 failed.
```

Dynamic programming usually uses arrays, which require less memory than maps based on hash tables.

Let's compare the run-times of all approaches. I only double the input four times (instead of the

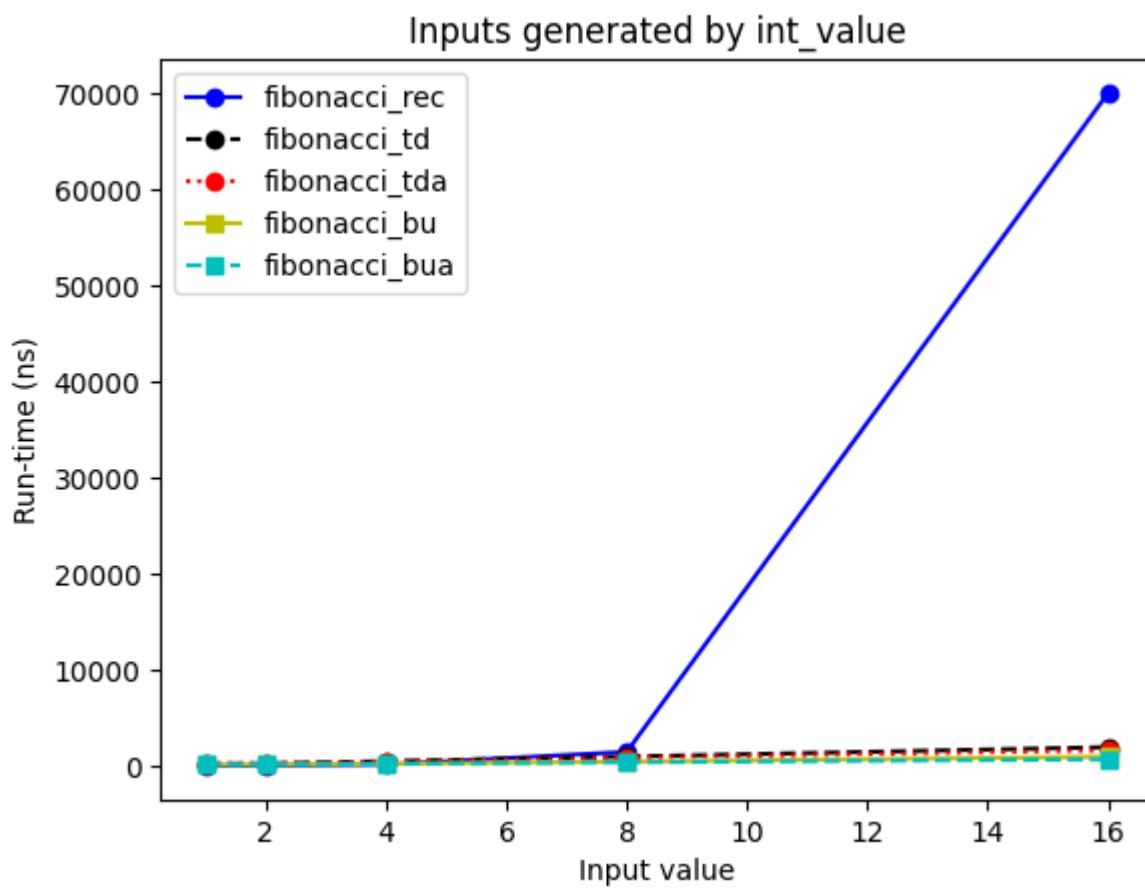
default ten times) because the recursive version takes too long (exponential time).

```
[10]: from algoesup import time_functions_int

time_functions_int(
    fibonacci_rec, fibonacci_td, fibonacci_tda, fibonacci_bu,
    ↪fibonacci_bua], double=4
)

Inputs generated by int_value

Input value      fibonacci_rec      fibonacci_td      fibonacci_tda      ↪
    ↪fibonacci_bu      fibonacci_bua
    1                  46.3            259.5           278.4
    ↪ 141.8             167.8 ns        263.4           282.8
    2                  50.6            171.4 ns        480.7
    ↪ 143.0             179.1 ns        509.9           880.5
    4                  244.0 ns         993.8           1640.2
    ↪ 239.0             395.2 ns
    8                  1483.4          1972.4
    ↪ 467.8             70093.6
    16                 708.5 ns
```



As expected, an array-based cache leads to shorter times than a dictionary-based cache. The

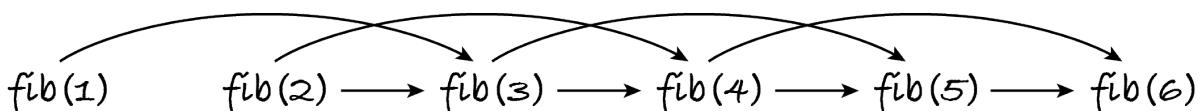
recursive version doesn't set up a cache and so it is initially the fastest, but the space-time tradeoff of the other versions quickly pays off, as the recursive version gets slower and slower.

23.1.5 A graph perspective

An alternative way of understanding dynamic programming is to look at the dependency graph of the problem instances. The nodes are the instances to be solved. If instance A must be solved before instance B, we draw an edge from A to B. The directed graph is acyclic because an instance can't depend on itself: that would make it impossible to solve.

Here's the DAG (directed acyclic graph) for $n = 6$. It's the recursive call tree with repeated nodes merged.

Figure 23.1.3



The DAG shows that each instance (except the base cases) depends on two smaller ones: every node, except for the base cases, has two in-neighbours.

Most nodes also have two out-neighbours, i.e. the solution of one instance is needed for two further instances. If there are edges from A to B and from A to C, this means instances B and C overlap because they have subproblem A in common. Hence we need dynamic programming, because recursion without a cache would repeatedly compute the common subproblems.



Note: If the DAG of the problem instances and their dependencies shows that instances overlap, i.e. there are nodes with a common in-neighbour, then use dynamic programming.

Looking at the DAG, we can see that both recursive versions (with and without cache) follow the edges backwards: they start with the right-most node (the input instance, here $n = 6$) and for each node do the two recursive calls corresponding to its in-neighbours.

Bottom-up dynamic programming instead starts with the left-most nodes (the base cases) and uses the subproblems solved so far to solve the next subproblem. It follows the dependency edges to solve increasingly larger subproblems until it reaches the input value (right-most node).

Dynamic programming caches the solution for a problem instance after solving it. This means its subproblems have been solved, and thus cached, beforehand. In other words, a subproblem P is cached after all subproblems that P depends on were cached. This means the cache is filled in topological order.



Note: Dynamic programming fills the cache (i.e. solves the subproblems) according to the topological sort of the instance dependency DAG.

The DAG is just a conceptual device to understand subproblem dependencies and in which order the subproblems are added to the cache. As you've seen in this section, dynamic programming algorithms fill the cache without creating a graph data structure and computing a topological sort.

23.2 Longest common subsequence

As explained in Sections 4.1.2 and 4.6.1, substrings and subsequences are zero or more items taken from a given sequence, without changing their order. A substring (also called a slice) contains consecutive items from the sequence, whereas a subsequence doesn't have to. Every substring is a subsequence, but not every subsequence is a substring. For example, (1, 2) is a substring and a subsequence of (1, 2, 3), whereas (1, 3) is a subsequence but not a substring, because 1 and 3 don't appear consecutively in (1, 2, 3).

This section further illustrates dynamic programming with the **longest common subsequence (LCS)** problem: given two sequences A and B, we want a longest of all sequences that are both a subsequence of A and a subsequence of B. For example, the LCS of (1, 3, 5, 7, 9, 11) and (2, 3, 5, 7, 11, 13) is (3, 5, 7, 11). Some sequences have several longest common subsequences, e.g. (1, 2, 3, 2) and (3, 2, 1, 2) have three LCS: (1, 2), (2, 2) and (3, 2). Our algorithm will return one of them.

The LCS is an indication of how similar two sequences are. For example, the Linux `diff` command (introduced in TM129) and word processors can compare two versions of a text file and show which lines were added and removed between the versions. They use quite sophisticated algorithms but one simple way to solve that problem is to determine which lines haven't changed, i.e. are common to both versions. Since new lines of text can be inserted between existing lines, the lines of the original file may not appear consecutively in the new file, so we must look for a subsequence, not a substring.

As an example, let the original file be the sequence of lines (A, B, C, D) and let the new file be (B, E, D, C, F). Each letter represents the content of one line. Line A was removed, lines C and D were swapped and lines E and F were added.

Every line of the original file that isn't in the LCS was removed and every line of the new file that isn't in the LCS was added, because those lines aren't in the common subsequence. Sequences (A, B, C, D) and (B, E, D, C, F) have two LCS: (B, C) and (B, D). Both show that line A was removed and lines E and F were added. If the output is (B, C) then this means line D was removed from after C and added before C. If the output is (B, D) then it's line C that is considered to have moved from before D to after D.

Another application of the LCS problem is in bioinformatics. We can represent a *DNA strand* as a string of the letters A, C, G and T. Computing the LCS (or some other similarity measure) of two strands allows scientists to find out if two genes have the same function, if one species evolved from another, if two people are related, etc.

In summary, the LCS and similar problems that involve comparing two sequences have many applications and they are often solved with dynamic programming.

In the rest of this section, to make the examples shorter to type, I will only use sequences of characters (strings). I'll call the two input strings *left* and *right*.

Here are some tests. Feel free to add your own. Make sure there's only one LCS for each test,

otherwise your algorithm may pick a different LCS and fail the test.

```
[1]: from algoesup import check_tests, test

DNA_LEFT = "A" * 6
DNA_RIGHT = "GATTACA" * 3 # more A's than DNA_LEFT

lcs_tests = [
    # case,           left,       right,      LCS
    ('one is empty', 'hello',    '',          ''),
    ('same string',   'hello',    'hello',    'hello'),
    ('nothing common', 'yes',     'no',        ''),
    ('typical case',  'soho',    'ohio',     'oho'),
    ('subsequence',   DNA_LEFT,  DNA_RIGHT,  DNA_LEFT),
    ('substring',     'TACAG',   DNA_RIGHT,  'TACAG')
]

check_tests(lcs_tests, [str, str, str])
OK: the test table passed the automatic checks.
```

23.2.1 Recursive

To obtain a dynamic programming solution, we must start with a recursive definition of the problem. At this point you may wish to read again [Section 12.8](#) and skim Sections [12.3](#) and [12.5](#) to remind yourself how sequences are processed recursively.

As usual, I start by thinking of the base cases. The smallest problem instances are always base cases because they can't be further decreased. The inputs of the LCS problem are strings, so the base cases are when either or both strings are empty. If one string is empty, it has no common characters with the other string, so their LCS is the empty string.

- if $left$ or $right$ is empty: $\text{lcs}(left, right) = "$

If neither string is empty, we can separate each one into a head (the first character) and a tail (the rest of the string). There are only two possible cases: their heads are the same or they differ.

If both strings have the same head, i.e. start with the same character, then that character is common to both and we must include it in the LCS. (If we ignored this common character, we wouldn't get the longest subsequence.) The LCS will be that character followed by the LCS of the tails, e.g. the LCS of 'hello' and 'hill' is 'h' followed by the LCS of 'ello' and 'ill'.

- if $\text{head}(left) = \text{head}(right)$: $\text{lcs}(left, right) = \text{head}(left) + \text{lcs}(\text{tail}(left), \text{tail}(right))$

Since both heads are equal, I could have used $\text{head}(right)$ instead of $\text{head}(left)$. The $+$ operator means concatenation in this context.

If the two heads are different, what should we do? Let's look at some examples. When computing the LCS of 'soho' and 'ohio', we must skip the 's' so that we can then match the first two letters of 'oho' and 'ohio'. We have $\text{lcs}(\text{'soho'}, \text{'ohio'}) = \text{lcs}(\text{'oho'}, \text{'ohio'}) = \text{'oho'}$. If we instead skip the initial 'o' of 'ohio', we would get the wrong answer $\text{lcs}(\text{'soho'}, \text{'ohio'}) = \text{lcs}(\text{'soho'}, \text{'hio'}) = \text{'ho'}$.

On the other hand, when computing the LCS of ‘AAA’ and ‘GATTACA’, which is ‘AAA’, we can’t skip any As in the left string as all have to be matched to the As in the right string. To sum up, if the two heads are different, sometimes we must skip the left head, sometimes the right head, in order to get the longest output.

How do we know which head to skip, without looking ahead to the rest of the strings? We don’t. When computing the *height of a tree* recursively, we didn’t know which subtree was taller, so we computed the height of both and took the height of the tallest. Here we must do the same.

- if $\text{head}(\text{left}) \neq \text{head}(\text{right})$: $\text{lcs}(\text{left}, \text{right}) = \text{longest of } \text{lcs}(\text{left}, \text{tail}(\text{right})) \text{ and } \text{lcs}(\text{tail}(\text{left}), \text{right})$

If skipping either head leads to common subsequences of the same length, then we can pick either of them. The resulting LCS will depend on which head is skipped, but it will always have the same maximum length. For example,

- $\text{lcs}(\text{'aba'}, \text{'baca'}) = \text{lcs}(\text{'ba'}, \text{'baca'}) = \text{'ba'}$ if we skip the left head
- $\text{lcs}(\text{'aba'}, \text{'baca'}) = \text{lcs}(\text{'aba'}, \text{'aca'}) = \text{'aa'}$ if we skip the right head

but both LCS have the maximal length of 2.

Exercise 23.2.1

Complete the following function and test it. The `m269_rec_list.py` file has functions `head` and `tail` that allow your code to closely follow the recursive definition. If you prefer, use `...[0]` and `...[1:]` instead.

[2]: %run -i ../m269_rec_list

```
def lcs(left: str, right: str) -> str:
    """Return the longest common subsequence of both strings."""
    # if one or both strings are empty:
    #     return the empty string
    # elif both heads are equal:
    #     return the head concatenated with the LCS of both tails
    # else:
    #     # compute the LCS when skipping the right head
    #     # compute the LCS when skipping the left head
    #     # return the longest of the two

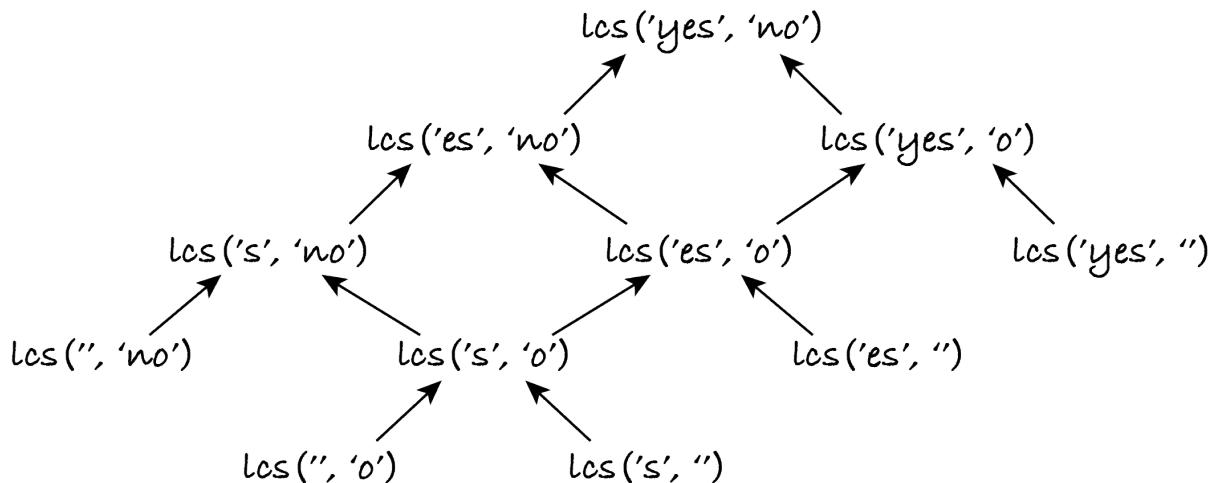
test(lcs, lcs_tests)
%timeit -r 3 lcs(DNA_LEFT, DNA_RIGHT)
```

Answer

23.2.2 Top-down

Before we invest effort in developing a dynamic programming solution, we should check whether it's worth it: are any subproblems repeatedly solved? Let's draw the DAG of the problem instances and their dependencies for input strings 'yes' and 'no'.

Figure 23.2.1



As the DAG shows, there are overlapping subproblems, e.g. $\text{lcs}(\text{'es'}, \text{'no'})$ and $\text{lcs}(\text{'yes'}, \text{'o'})$ overlap on common subproblem $\text{lcs}(\text{'es'}, \text{'o'})$. Common subproblems arise from the multiple ways to reach the same tails of the initial input strings, by skipping characters from the left and right strings a in different order. For example, $\text{lcs}(\text{'es'}, \text{'o'})$ is solved twice: first by skipping 'y' then 'n', and second by skipping 'n' then 'y'.

This means that instance ('s', 'o') is solved three times:

- $\text{lcs}(\text{'yes'}, \text{'no'}) \leftarrow \text{lcs}(\text{'es'}, \text{'no'}) \leftarrow \text{lcs}(\text{'es'}, \text{'o'}) \leftarrow \text{lcs}(\text{'s'}, \text{'o'})$
- $\text{lcs}(\text{'yes'}, \text{'no'}) \leftarrow \text{lcs}(\text{'yes'}, \text{'o'}) \leftarrow \text{lcs}(\text{'es'}, \text{'o'}) \leftarrow \text{lcs}(\text{'s'}, \text{'o'})$
- $\text{lcs}(\text{'yes'}, \text{'no'}) \leftarrow \text{lcs}(\text{'es'}, \text{'no'}) \leftarrow \text{lcs}(\text{'s'}, \text{'no'}) \leftarrow \text{lcs}(\text{'s'}, \text{'o'})$

This in turn means that both leaf instances in the bottom row are also solved three times each. More generally, the number of times an instance is solved is the number of different paths from ('yes', 'no') to it. (Remember that recursion follows the DAG edges backwards.)

Caching the results for subproblems avoids repeated recursive calls. For example, the second time $\text{lcs}(\text{'es'}, \text{'o'})$ is called, the cache is looked up and no calls to $\text{lcs}(\text{'s'}, \text{'o'})$ and $\text{lcs}(\text{'es'}, \text{''})$ are made.

Exercise 23.2.2

Paste a copy of your code for the previous exercise into the auxiliary function below and modify it to use the `cache` dictionary.

[3]: %run -i ../m269_rec_list

```
def lcs_topdown(left: str, right: str) -> str:
```

(continues on next page)

(continued from previous page)

```

"""Return the LCS of both strings using top-down dynamic
programming."""

def lcs(left: str, right: str) -> str:
    """Auxiliary recursive function."""
    # if problem instance (left, right) isn't in cache:
    # compute LCS recursively and store it in cache
    # optional: print the cached value to see how the cache
    is filled
    # return the cached LCS for left and right

    cache = dict()
    return lcs(left, right)

```

If you add a print statement I recommend you only run

```
[4]: lcs_topdown("yes", "no")
```

and check the order in which the solutions are cached against the DAG above. Then comment out the print statement and run the next cell. The run-time should be lower than for the original recursive version.

```
[5]: test(lcs_topdown, lcs_tests)
%timeit -r 3 lcs_topdown(DNA_LEFT, DNA_RIGHT)
```

Hint Answer

23.2.3 Recursive with indices

Before proceeding to the bottom-up version, let's improve the efficiency of what we have. As noted in [Section 12.6](#), we should avoid slicing in every recursive call and use indices instead.

For this problem two indices suffice, each pointing to the current head. I will use single-letter names l and r for the indices, because more descriptive names like *left index* make the recursive definition too verbose and harder to read, in my opinion.

Function $\text{lcs}(l, r)$ will compute the LCS of the *left* string from index l onwards and of the *right* string from index r onwards. To obtain the LCS of both strings from the start, we compute $\text{lcs}(0, 0)$.

The bases cases are when there's nothing more to process in either string, which happens when either index reaches the end of the corresponding string:

- if $l = |\text{left}|$: $\text{lcs}(l, r) = "$
- if $r = |\text{right}|$: $\text{lcs}(l, r) = "$

The recurrence relations remain the same, just written differently. The head of a string is the character at position l or r . The tail of a string comprises the characters from the next position: $l+1$ or $r+1$.

- if $left[l] = right[r]$: $\text{lcs}(l, r) = left[l] + \text{lcs}(l+1, r+1)$
- otherwise: $\text{lcs}(l, r) = \text{longest of } \text{lcs}(l, r+1) \text{ and } \text{lcs}(l+1, r)$

Exercise 23.2.3

Implement the new recursive definition and run it. The run-time should be lower than in Exercise 23.2.1 (with slicing) and higher than in Exercise 23.2.2 (with cache).

```
[6]: def lcs_indices(left: str, right: str) -> str:
    """Return the LCS of left and right using indices, not slicing."""
    ↵

    def lcs(l: int, r: int) -> str:
        """Return the LCS of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        """
        pass

    return lcs(0, 0)

test(lcs_indices, lcs_tests)
%timeit lcs_indices(DNA_LEFT, DNA_RIGHT)
```

Answer

23.2.4 Top-down with matrix

Like for the *Fibonacci problem*, we can use arrays instead of hash tables. By changing the inputs of the recursive function from strings to integer indices, the cache can be implemented as a matrix: `cache[1][r]` is the LCS for strings `left[1:]` and `right[r:]`. Since l and r start at zero and go up to the length of the strings, the matrix has $|left| + 1$ rows and $|right| + 1$ columns.

To understand how the matrix cells depend on each other, we must look at the recurrence relations. Here they are again, with the irrelevant info left out:

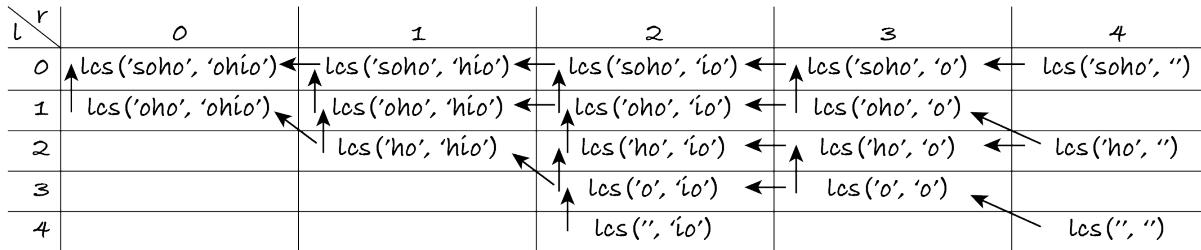
- if ...: $\text{lcs}(l, r) = \dots \text{lcs}(l+1, r+1)$
- ...: $\text{lcs}(l, r) = \dots \text{lcs}(l, r+1) \text{ and } \text{lcs}(l+1, r)$

From this we can see that `cache[1][r]`, which is $\text{lcs}(l, r)$, may depend on `cache[1+1][r+1]` or on `cache[1][r+1]` and `cache[1+1][r]`. The value of the matrix cell at row l and column r depends on values that are in the next row $l + 1$, or in the same row l but in the next column $r + 1$.

Here's the DAG of dependencies for inputs 'soho' and 'ohio', shown as it's stored in the matrix. As we go down the rows (l increases), we skip letters in the left string; as we go right in the columns (r increases), we skip letters in the right string. If the letters at indices l and r are the same, the algorithm skips that letter in both strings, and so those cells depend on the cells diagonally below: `cache[1+1][r+1]`.

If instead the letters are different, the cell depends on two other cells. For example, the value in row 0, column 2, is the LCS of ‘soho’ (no letters skipped in the left string) and ‘io’ (two letters skipped in the right string). The LCS in that cell is the longest of the LCS in the two cells immediately below and to the right (`cache[1+1][r]` and `cache[1][r+1]`) because ‘soho’ and ‘io’ don’t start with the same letter.

Figure 23.2.2



The empty cells are never filled by the top-down algorithm. Which cells are left empty depends on the input strings, but the recursive calls always start in the top left corner (row 0, column 0). Each call either makes one further recursive call to fill the cell diagonally below or makes two recursive calls to fill the cells below and to the right. The cells in the last row and column store the LCS of the base cases: at least one string is empty.

Now that you’ve seen how the top-down algorithm works, implement it.

Exercise 23.2.4

Copy your `lcs` auxiliary function from Exercise 23.2.3 and add a matrix cache. I’ve initialised the matrix for you with `None`. We can’t use the empty string as that’s a valid LCS value.

```
[7]: def lcs_topdown_matrix(left: str, right: str) -> str:
    """Return the LCS of both strings using top-down dynamic
    programming."""
    def lcs(l: int, r: int) -> str:
        """Return the LCS of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        """
        # if lcs(l, r) isn't in cache:
        #     compute it recursively and store it in cache
        #     optional: print the cached value
        # return the cached lcs(l, r)

        cache = []
        for row in range(len(left) + 1):  # noqa: B007
            cache.append([None] * (len(right) + 1))
        return lcs(0, 0)
```

If you add a print statement I recommend you only run

```
[8]: lcs_topdown_matrix("soho", "ohio")
```

and check the order in which the matrix is filled against Figure 23.2.2 to confirm it follows a topological sort. Then comment out or remove the print statement and run the next cell. You should obtain the lowest run-time so far.

```
[9]: test(lcs_topdown_matrix, lcs_tests)
%timeit lcs_topdown_matrix(DNA_LEFT, DNA_RIGHT)
```

Hint Answer

23.2.5 Bottom-up

As we've seen above, when describing the top-down approach, each cell in the matrix either depends on

- zero cells, if it's a base case (bottom row and right-most column)
- one cell diagonally below to the right, if both substrings start with the same letter
- or two cells to the right and below, if the substrings start with different letters.

The next figure shows the completely filled matrix. The shaded cells are the cells that the top-down algorithm didn't fill because they don't contribute to solving $\text{lcs}(\text{'soho'}, \text{'ohio'})$: there's no path of dependencies from those cells to the top left cell.

Figure 23.2.3

$l \setminus r$	0	1	2	3	4
0	$\text{lcs}(\text{'soho'}, \text{'ohio'})$	$\text{lcs}(\text{'soho'}, \text{'hio'})$	$\text{lcs}(\text{'soho'}, \text{'io'})$	$\text{lcs}(\text{'soho'}, \text{'o'})$	$\text{lcs}(\text{'soho'}, \text{''})$
1	$\text{lcs}(\text{'oho'}, \text{'ohio'})$	$\text{lcs}(\text{'oho'}, \text{'hio'})$	$\text{lcs}(\text{'oho'}, \text{'io'})$	$\text{lcs}(\text{'oho'}, \text{'o'})$	$\text{lcs}(\text{'oho'}, \text{''})$
2	$\text{lcs}(\text{'ho'}, \text{'ohio'})$	$\text{lcs}(\text{'ho'}, \text{'hio'})$	$\text{lcs}(\text{'ho'}, \text{'io'})$	$\text{lcs}(\text{'ho'}, \text{'o'})$	$\text{lcs}(\text{'ho'}, \text{''})$
3	$\text{lcs}(\text{'o'}, \text{'ohio'})$	$\text{lcs}(\text{'o'}, \text{'hio'})$	$\text{lcs}(\text{'o'}, \text{'io'})$	$\text{lcs}(\text{'o'}, \text{'o'})$	$\text{lcs}(\text{'o'}, \text{''})$
4	$\text{lcs}(\text{''}, \text{'ohio'})$	$\text{lcs}(\text{''}, \text{'hio'})$	$\text{lcs}(\text{''}, \text{'io'})$	$\text{lcs}(\text{''}, \text{'o'})$	$\text{lcs}(\text{''}, \text{''})$

In the bottom-up approach we need to fill the matrix so that when a cell is computed, the cells it depends on are already filled. Since a cell depends on cells below and to the right, this means we must fill the cache from the last to the first row and, within each row, from the last to the first column.

```
[10]: def lcs_bottomup(left: str, right: str) -> str:
    """Return the LCS of both strings using bottom-up dynamic programming."""
    # create cache as in top-down approach
    cache = []
    for row in range(len(left) + 1): # noqa: B007
        cache.append([None] * (len(right) + 1))

    # compute LCS bottom-up
    for l in range(len(left), -1, -1): # last to first row
        for r in range(len(right), -1, -1): # last to first column
            if left[l] == right[r]:
                cache[l][r] = cache[l+1][r+1] + left[l]
            else:
                cache[l][r] = max(cache[l+1][r], cache[l][r+1])
```

(continues on next page)

(continued from previous page)

```

for r in range(len(right), -1, -1): # last to first column
    if l == len(left) or r == len(right):
        cache[l][r] = ""
    elif left[l] == right[r]:
        cache[l][r] = left[l] + cache[l + 1][r + 1]
    else:
        skip_left = cache[l + 1][r]
        skip_right = cache[l][r + 1]
        if len(skip_left) > len(skip_right):
            cache[l][r] = skip_left
        else:
            cache[l][r] = skip_right

# change the next line to see the contents of the matrix for
→other tests
if left == "soho" and right == "ohio":
    for l in range(len(left) + 1):
        print(cache[l])

# solution is in top left corner of matrix (l = r = 0)
return cache[0][0]

```

```

test(lcs_bottomup, lcs_tests)
%timeit lcs_bottomup(DNA_LEFT, DNA_RIGHT)

```

```

Testing lcs_bottomup...
['oho', 'ho', 'o', 'o', '']
['oho', 'ho', 'o', 'o', '']
['ho', 'ho', 'o', 'o', '']
['o', 'o', 'o', 'o', '']
['', '', '', '', '']
Tests finished: 6 passed (100%), 0 failed.
16.3 µs ± 12 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops
→each)

```

23.2.6 Complexity and run-time

The complexity of dynamic programming is the number of entries in the cache multiplied by the complexity of computing each entry. The bottom-up approach always computes all entries, while top-down only computes those necessary to solve the given problem instance. In the worst case, the top-down recursive calls also fill the whole cache, so the bottom-up complexity is the same as the top-down worst-case complexity.

For the LCS problem, the matrix has $(|left| + 1) \times (|right| + 1)$ cells but for complexity purposes we can ignore the $+1$. Each cell takes constant time to compute if it's the longest of the strings in the cells below and to the right, or it takes linear time if it's the concatenation of the head with

the string in the cell diagonally below.

The LCS of two strings can never be longer than the shortest string, because it only includes common characters. So the concatenation takes $\Theta(\min(|left|, |right|))$ in the worst case.

To sum up, the LCS problem can be solved with dynamic programming in $\Theta(\min(|left|, |right|) \times |left| \times |right|)$ time.

When I ran the various solutions I obtained roughly these times for the LCS of `DNA_LEFT` and `DNA_RIGHT`:

- recursive: 140 µs with slicing and 80 µs without
- top-down: 55 µs with slicing and 30 µs without
- bottom-up: 47 µs.

Like for the ‘soho’ and ‘ohio’ example, the bottom-up approach solves all 7×22 subproblems of `lcs('AAAAAA', 'GATTACAGATTACAGATTACA')`, whereas the top-down approach only solves the necessary ones, so it is faster in spite of the overhead of recursive calls.



Note: When solving a problem with dynamic programming, you may have to implement both the top-down and bottom-up approaches to see which is faster for your typical problem instances.

23.3 Knapsack

The 0/1 knapsack problem asks for the most valuable subset of items that doesn’t exceed a given weight limit. The problem can be solved with *exhaustive search* or *backtracking* but not with *greed*. Let’s now consider dynamic programming.

Bottom-up dynamic programming is usually implemented with for-loops iterating over an index-based cache. Sets can’t be indexed, but sequences can. So let’s slightly modify the formulation of the problem, using sequences and subsequences instead of sets and subsets.

Function: `knapsack`

Inputs: `items`, a sequence of pairs of integers; `capacity`, an integer

Preconditions: for each $(weight, value)$ in `items`, $weight > 0$ and $value > 0$; $capacity \geq 0$

Output: `packed`, a sequence of pairs of integers

Postconditions:

- `packed` is a subsequence of `items`
- the sum of the weights in `packed` doesn’t exceed `capacity`
- the sum of the values in `packed` is largest among all sequences satisfying the previous two conditions

Here are the necessary constants and functions and some tests.

```
[1]: from algoesup import check_tests, test

WEIGHT = 0
VALUE = 1


def weight(items: list) -> int:
    """Return the total weight of the items."""
    total = 0
    for item in items:
        total = total + item[WEIGHT]
    return total


def value(items: list) -> int:
    """Return the total value of the items."""
    total = 0
    for item in items:
        total = total + item[VALUE]
    return total


ITEMS = [(2, 3), (2, 4), (3, 4), (4, 20), (5, 30)]
knapsack_tests = [
    # case,           items,   capacity,      knapsack
    ('none fits',    ITEMS,    1,              []),
    ('all fit',      ITEMS,    16,             ITEMS),
    ('one is best',  ITEMS,    6,              [(5, 30)])
]

check_tests(knapsack_tests, [list, int, list])
OK: the test table passed the automatic checks.
```

23.3.1 Recursive

We must first recursively define the function we want to compute, namely $\text{knapsack}(items, capacity)$.

As usual, we start with the bases cases. The two smallest possible inputs correspond to no items and no capacity. In both cases nothing can be put in the knapsack: the output is the empty sequence.

- if $items = ()$: $\text{knapsack}(items, capacity) = ()$
- if $capacity = 0$: $\text{knapsack}(items, capacity) = ()$

If $items$ isn't the empty sequence, we can separate it into a head (the next item to consider) and a tail (the other items). Like for the backtracking solution, there are only two cases: either the

head item fits in the knapsack or it doesn't. If the latter, i.e. the item weighs more than the remaining capacity, it must be skipped.

- if $\text{weight}(\text{head}(\text{items})) > \text{capacity}$: $\text{knapsack}(\text{items}, \text{capacity}) = \text{knapsack}(\text{tail}(\text{items}), \text{capacity})$

If the item fits, we don't know whether we should put it in the knapsack or not, so we try both options and choose the one that leads to the most valuable knapsack. To make the final recurrence relation less verbose and easier to read, I use *item* as an abbreviation of $\text{head}(\text{items})$.

- otherwise: $\text{knapsack}(\text{items}, \text{capacity}) = \text{most valuable of}$
 - $\text{knapsack}(\text{tail}(\text{items}), \text{capacity})$ and
 - $(\text{item}) + \text{knapsack}(\text{tail}(\text{items}), \text{capacity} - \text{weight}(\text{item}))$

The + operator is again the concatenation operator. If the item is put in the knapsack, the remaining capacity is decreased. Here's my implementation.

```
[2]: def knapsack(items: list, capacity: int) -> list:  
    """Return a highest-value subsequence of items that weigh at  
    most capacity.  
  
    Preconditions:  
    - items is a list of weight-value pairs, both positive integers  
    - capacity ≥ 0  
    """  
  
    if len(items) == 0 or capacity == 0:  
        return []  
    else:  
        item = items[0] # head  
        rest = items[1:] # tail  
        skip = knapsack(rest, capacity)  
        # if item doesn't fit, we must skip it  
        if item[WEIGHT] > capacity:  
            return skip  
        # otherwise take it if that leads to a higher value  
        else:  
            take = [item] + knapsack(rest, capacity - item[WEIGHT])  
            if value(skip) > value(take):  
                return skip  
            else:  
                return take  
  
test(knapsack, knapsack_tests)
```

Testing knapsack...
Tests finished: 3 passed (100%), 0 failed.

The next step, in preparation for dynamic programming, is to use indices.

Exercise 23.3.1

Copy the code above into the auxiliary function below and modify it so that it uses the `index` argument instead of repeatedly slicing the `items` list. Add the call to the auxiliary function.

```
[3]: def knapsack_indices(items: list, capacity: int) -> list: # noqa:
    ↪D103
    # docstring not repeated

    def knapsack(index: int, capacity: int) -> list:
        """Return a subsequence of items[index:].

        Preconditions: 0 ≤ index ≤ len(items) and 0 ≤ capacity
        Postconditions: the output fits the capacity and maximises
        ↪the value
        """

        pass

    pass # call the auxiliary function and return the solution

test(knapsack_indices, knapsack_tests)
```

Hint Answer

23.3.2 Common subproblems

Like for the Fibonacci and LCS problems, the next step is to think whether there are common subproblems that would benefit from caching the solutions.

Is it possible for the recursive function `knapsack(items, capacity)` or `knapsack(index, capacity)` to be called several times with the exact same input values? (Hint: consider the list of items `[(1, 3), (1, 4), (3, 4), (4, 20)]`.)

In the LCS problem, skipping first the left head and then the right head or vice versa leads to the same subproblem. Here we can have the same issue. If two items have the same weight, like the two items with weight 1 in the hint, then whether we skip the first and take the second or take the first and skip the second, we will arrive at the same subproblem, with the same remaining items and capacity. For the example in the hint, the recursive algorithm twice solves the subproblem `knapsack([(3, 4), (4, 20)], capacity - 1)`, or `knapsack(2, capacity - 1)` in the version with indices. This means that any further subproblem will also be solved at least twice. Twice the item with weight 3 will be taken to solve `knapsack([(4, 20)], capacity - 4)` and twice that item will be skipped to solve `knapsack([(4, 20)], capacity - 1)`, and so on. If we draw the DAG we can easily see how many paths lead to each subproblem.

Exercise 23.3.2

1. Draw the DAG of the subproblems of knapsack([(1, 3), (1, 4), (3, 4), (4, 20)], 4).

To make the DAG less tedious to write, I suggest you omit ‘knapsack’ and just write a list–capacity pair or, even shorter, an index–capacity pair. Here are three of the DAG’s nodes:

Subproblem	Shorthand	Notes
([(1, 3), (1, 4), (3, 4), (4, 20)], 4)	(0, 4)	no item processed, capacity is 4
([(1, 4), (3, 4), (4, 20)], 4)	(1, 4)	first item skipped, capacity still 4
([(1, 4), (3, 4), (4, 20)], 3)	(1, 3)	first item added, reducing capacity by 1

2. After drawing the DAG, find which subproblems are solved more than once.

Hint Answer

23.3.3 Top-down and bottom-up

The next step is to think about the cache, which is the same for the top-down and bottom-up approaches. As always, the cache stores the solution of each problem instance. For this problem, the cache stores lists of items, namely the most valuable knapsack for each problem instance, which is a pair of integers: the current index and the remaining capacity.

What’s the best way to implement the cache? With a hash table (Python dictionary), an array or a matrix? If it’s a hash table, what are the keys? If it’s an array or matrix, what do the indices represent and how many are there, i.e. what is the size of the array or matrix?

Like the LCS problem, this one has two inputs that are natural numbers and can be used as indices to look up the solution in a matrix: `cache[i][c]` is the solution for instance `knapsack(i, c)`, i.e. it’s the most valuable subsequence of `items[i:]` that doesn’t exceed weight `c`.

The row index goes from 0 to `|items|` and is the index of the current item. The column index goes from 0 to `capacity` and indicates the remaining capacity.

I could instead use columns for item indices and rows for capacities, but I find `cache[i][c]` more intuitive than `cache[c][i]` because in the former the matrix indices follow the same order as the problem inputs.

Exercise 23.3.3

Copy your code for Exercise 23.3.1 and modify it so that it creates and uses a cache in a top-down fashion.

```
[4]: def knapsack_topdown(items: list, capacity: int) -> list: # noqa: D103
    def knapsack(index: int, capacity: int) -> list:
        """Return a subsequence of items[index:].

```

Preconditions: $0 \leq index \leq len(items)$ and $0 \leq capacity$

(continues on next page)

(continued from previous page)

```

Postconditions: the output fits the capacity and maximises_
→the value
    """
pass

pass # create an empty cache and call the auxiliary function

```

If you add a print statement to trace how the cache is filled, run

[5]: knapsack_topdown([(1, 3), (1, 4), (3, 4), (4, 20)], 4)

and check the caching follows a topological sort of the DAG for *Exercise 23.3.2*.

Afterwards, uncomment your print statement before running all tests.

[6]: test(knapsack_topdown, knapsack_tests)

Hint Answer

The next step is to think in which order to fill the cache for the bottom-up approach. Here again are the recurrence relations, with the parts that don't relate to subproblem dependencies omitted:

- if ...: knapsack(*items*, *capacity*) = knapsack(tail(*items*), *capacity*)
- otherwise: knapsack(*items*, *capacity*) = ... of
 - knapsack(tail(*items*), *capacity*) and
 - ... + knapsack(tail(*items*), *capacity* - weight(*item*))

If we convert the above to matrix cells, which cells does `cache[i][c]` possibly depend on? (Hint: reformulate the recurrence relations using indices.)

The recurrence relations (and the code) tell us that $\text{knapsack}(i, c)$ depends on $\text{knapsack}(i+1, c)$ and $\text{knapsack}(i+1, c-w)$, where w is the weight of $\text{items}[i]$. This means that each matrix cell depends on cells in the next row and in columns to the left (lower-capacity value).

Therefore, in which order should the rows of the matrix be filled? And in which order should the columns be filled?

The rows must be filled from last to first. As for the columns, they must be filled first to last.

Exercise 23.3.4

Copy your top-down dynamic programming approach to the next cell and modify it so that it fills the cache iteratively.

```
[7]: def knapsack_bottomup(items: list, capacity: int) -> list: # noqa:  
    ↪D103  
    # create the cache  
    # for each row from last to first:  
    # for each column:  
        # compute cache[row][column]  
    pass # return the cell with the solution for items and capacity  
  
test(knapsack_bottomup, knapsack_tests)
```

Hint Answer

If you haven't attempted the exercises, I suggest you copy my solutions to this notebook, so that they are next to each other. That way you will see that the recursive, top-down and bottom-up algorithms are mostly the same, and that each one can be systematically derived from the previous one.

23.3.4 Complexity

As always, the worst-case top-down and bottom-up complexities are the same: the matrix size multiplied by the worst-case complexity of filling each cell.

What's the size of the matrix, in terms of $|items|$ and $capacity$? What's the worst-case complexity of filling each cell? (Hint: the reasoning is similar to the LCS problem.)

The matrix has $(|items| + 1) \times (capacity + 1)$ cells. In the worst case, a cell requires

- concatenating the head item with the best knapsack for the tail and remaining capacity
- comparing the values of two knapsacks (with and without the item).

A knapsack can never have more items than the input list, so the worst-case complexity for computing the value of a cell is $\Theta(|items|)$. The overall worst-case complexity is $(|items| + 1) \times (capacity + 1) \times \Theta(|items|) = \Theta(|items|^2 \times capacity)$.

This is usually much better than the exponential complexity for generating and testing all subsequences of $items$. However, as the capacity approaches 2^n , with $n = |items|$, the dynamic approach may not be much faster. For example, if we have ten items to put in a container with 1000 kg capacity, then the cache has over ten thousand cells. Filling them all in the bottom-up approach may take longer than generating the $2^{10} = 1024$ subsets of ten items.

23.4 Summary

Dynamic programming is a technique that makes a space–time tradeoff to more efficiently solve problems that have **overlapping subproblems**: problem instances that depend on a common subproblem (a smaller problem instance). A pure recursive algorithm solves each common subproblem (and the subproblems it depends on) several times. Dynamic programming uses a **cache**: a map of subproblems to their solutions. When a subproblem is first solved, its

solution is stored in the cache so that it can be subsequently looked up instead of having to be recomputed.

Top-down dynamic programming stores the solutions to subproblems as they're recursively solved. At the end, the cache only contains the solutions to those subproblems that are needed to solve the initial problem instance given as input.

Bottom-up dynamic programming iteratively solves all subproblems, from the smallest to the largest (the initial problem instance).

If we represent subproblems and their dependencies as a directed acyclic graph, then the common subproblems are those with two or more out-neighbours. Dynamic programming fills the cache in a topological order of the graph, so that each subproblem P is solved by using the already cached solutions to the subproblems that P depends on.

The worst-case complexity of top-down and bottom-up dynamic programming is the maximal size of the cache (number of subproblems) multiplied by the worst-case complexity to solve one subproblem.

CHAPTER 24

PRACTICE 3

This chapter introduces no new concepts, only new problems to practise algorithmic techniques and data structures. You should skim the advice in the *next chapter* before attempting the problems. They are in increasing difficulty order, in my opinion, but your mileage may vary.

Each problem can be solved with a technique or algorithm introduced in Chapters 21 to 23, possibly in addition to earlier algorithms and data structures.

The advice of *Chapter 9* applies again:

- Read all problems first, to decide in which order to tackle them.
- Don't worry if you don't finish them all: next week's TMA preparation has priority.
- Compare approaches with your study buddy.
- Post alternative solutions to mine in the forums, with appropriate spoiler alerts.

This chapter's problems support the following learning outcomes.

- Develop and apply algorithms and data structures to solve computational problems – you will apply several of the ADTs and techniques you learned.
- Explain how an algorithm or data structure works, in order to communicate with relevant stakeholders – some exercises ask you to only outline the algorithm.
- Write readable, tested, documented and efficient Python code – you will adapt generic code templates to the problem at hand and use Python data types and operations.

If you have the time and interest in solving further problems, have a look at Steven Halim's [Methods to Solve](#) website, which provides one-line hints to thousands of problems, although most require techniques not covered in M269. I selected several problems in this book from there.

Another good site for practising is [LeetCode](#) as it's easy to search for problems on a particular topic. All problems have official or user-provided solutions in different languages.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

24.1 Safe places

To analyse the safety of a building in case it needs to be evacuated, the places and escape routes of the building have been modelled as a weighted directed graph. Each node is a place (office, meeting room, storage room, etc.) and each edge is a passage between places (corridor, stairs, door, etc.). The weights indicate the expected time to walk across that passage.

One node represents the space in front of the building, where staff and visitors must assemble. The assembly node can be reached from every other node.

Given a time duration, e.g. 10 minutes, we want to know from how many places it is possible to reach the assembly place within that duration.

24.1.1 Exercises

Exercise 24.1.1

M269 introduced these standard graph problems, in this order: shortest tour, shortest paths, minimum spanning trees, connected components, topological sorts, detecting cycles. Which one does this problem require you to solve?

Answer

Exercise 24.1.2

Complete the problem definition. The postconditions should refer to one or more standard graph problems.

Function: safe places

Inputs: *building*, a digraph with integer weights; *assembly*, an object; *time*, an integer

Preconditions:

- *assembly* is a node of *building*
- the weights and *time* are in the same unit (e.g. seconds) and aren't negative

Output: *safe*, an integer

Postconditions: *safe* is the number of nodes in *building* (except *assembly*) that ...

Answer

Exercise 24.1.3

Which of the graph algorithm(s) you learned must you use or adapt for this problem?

Answer

Exercise 24.1.4

Outline an exhaustive search algorithm that solves the problem.

Answer

Exercise 24.1.5

What's the worst-case complexity of your algorithm?

Hint Answer

Exercise 24.1.6

Outline a more efficient algorithm than exhaustive search to solve this problem.

Hint Answer

Exercise 24.1.7

Indicate the worst-case complexity of your algorithm for the previous exercise.

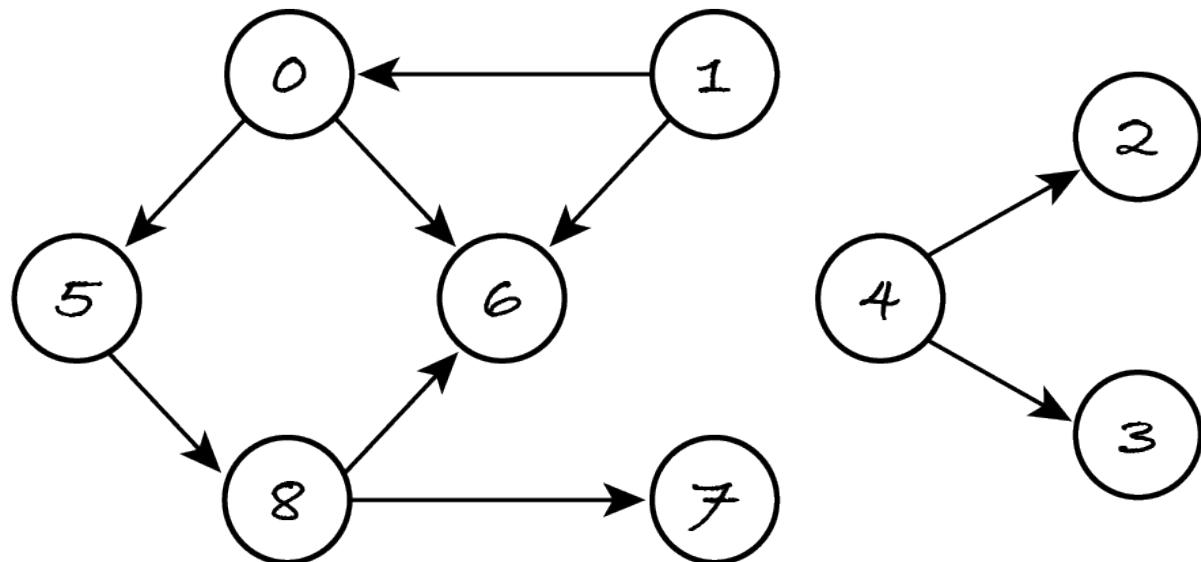
Answer

24.2 Extra staff

A project manager has constructed a directed acyclic graph (DAG) where the nodes represent the project's tasks and each edge A → B states that task A must be finished before task B starts. Each task takes one person one week to finish.

To shorten the duration of the project, the manager wants as many tasks as possible to be done simultaneously. Consider the following example.

Figure 24.2.1



This project takes five weeks to complete:

1. Tasks 1 and 4
2. Tasks 0, 2 and 3
3. Task 5
4. Task 8
5. Tasks 6 and 7.

The project team has P people. To budget for the project, the manager wants to know how many weeks require extra people. For the example, if $P = 2$, then the answer is one: only the second week needs extra people. But if $P = 1$, then the answer is three: the first, second and fifth weeks need extra people. Results for other values of P are given below, in the test table.

24.2.1 Exercises

Exercise 24.2.1

1. Which *standard graph problem* is relevant for this problem?
2. What algorithm will you need to use or adapt?

Answer

Exercise 24.2.2

Outline an algorithm to solve the problem. Indicate any changes you need to make to the algorithm in your previous answer.

Hint Answer

Exercise 24.2.3

Implement the algorithm.

```
[1]: %run -i ../m269_digraph
from algoesup import test

def extra_staff(project: DiGraph, people: int) -> int:
    """Return how many weeks need more than people on the project.

    Preconditions: project is acyclic, people ≥ 0
    """
    pass

project = DiGraph()
for task in range(9):
    project.add_node(task)
project.add_edge(1, 0)
```

(continues on next page)

(continued from previous page)

```

project.add_edge(1, 6)
project.add_edge(0, 5)
project.add_edge(0, 6)
project.add_edge(5, 8)
project.add_edge(8, 6)
project.add_edge(8, 7)
project.add_edge(4, 2)
project.add_edge(4, 3)

extra_staff_tests = [
    # case,      project, people, weeks
    ("0 people", project, 0,      5),
    ("1 person", project, 1,      3),
    ("2 people", project, 2,      1),
    ("3 people", project, 3,      0),
]

test(extra_staff, extra_staff_tests)
    
```

Hint Answer

24.3 Borrow a book

A group of friends goes to the same places every week. If one friend wants to borrow a book from another friend, the book is passed on when two friends meet at the same place on the same day.

Here's an example weekly schedule of three friends. Each person is at most in one place each day.

Person	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	gym		uni			pool	
Bob		pool	uni	uni	gym		gym
Celia	uni	pool	gym	pool	uni	pool	

If Alice wants to lend a book to Celia, Alice can pass the book to Bob on Wednesday when they're both at uni, then Bob passes it to Celia on Tuesday when they're both at the pool. If we start counting on Monday, it takes eight days (from Monday to Tuesday the week after) for Celia to get the book.

However, Alice can lend the book directly to Celia on Saturday, at the pool. That way Celia only waits five days (Monday to Saturday) to get the book.

Given the weekly schedule, the lender and the borrower, we want to know the fewest days the borrower waits to get the book, starting from a Monday. The number of days should be infinite if the book can't be passed on from the borrower to the lender.

24.3.1 Exercises

Exercise 24.3.1

Chapters 21 to 23 have introduced

- graph problems (finding components, computing a topological sort, modelling state transitions)
 - backtracking to solve constraint satisfaction and optimisation problems
 - dynamic programming to solve problems with overlapping subproblems.

Which approach do you think is needed for this problem and why?

Hint Answer

Exercise 24.3.2

Outline an algorithm and the required ADT or data structures.

Hint Answer

Exercise 24.3.3 (optional)

Implement and test your approach.

Hint

24.4 Levenshtein distance

When spell checkers detect a misspelled word, they only suggest similar words as replacements. They do this by computing the edit distance between a valid word and the misspelled word: the number of editing operations it takes to turn the misspelled word into the valid word. If the number of edits is low, e.g. three or less, then the valid word is added to the suggestions.

Different edit distances can be defined by allowing different kinds of operations. If the allowed operations are insertion, deletion and replacement, then the edit distance is commonly known as the Levenshtein distance. For example, ‘fast’ becomes ‘haste’ in two operations: replace ‘f’ with ‘h’, and insert ‘e’. Vice versa, deleting ‘e’ and replacing ‘h’ with ‘f’ turns ‘haste’ into ‘fast’. It’s not possible to transform one string into the other with fewer operations, so the Levenshtein distance of ‘fast’ and ‘haste’ is 2.

The Levenshtein distance can be computed for any two strings: the tests below include DNA strings and English words without misspellings. We'll name the function `edit(left, right)` instead of `levenshtein(left, right)` to make it faster and easier to type. The tests are:

```
[1]: from algoesup import check_tests, test

edit_tests = [
    # case, left,
    ('same word', 'hello'),
```

(continues on next page)

(continued from previous page)

```

('insert',           'rate',          'grate',        1),
('delete',          'rate',          'ate',          1),
('replace',         'rate',          'fate',         1),
('replace',         'algorithm',    'logarithm',   3),
('replace & delete', 'yes',          'no',          3),
('delete & insert', 'great',        'grate',       2),
('replace & insert', 'fast',          'haste',       2),
('all three edits', 'GATACA',       'CATCAT',      3),
# common typo: pressing neighbouring letter on keyboard
('replace',         'mimt',          'mint',         1),
# common typo: pressing neighbouring letters in wrong order
('swap letters',   'perquel',       'prequel',     2),
]

check_tests(edit_tests, [str, str, int])
OK: the test table passed the automatic checks.
    
```

24.4.1 Exercises

Exercise 24.4.1

Write a recursive definition of the Levenshtein distance, using the head and tail operations on strings. This is not trivial so break it down and think of concrete examples:

- What are the base cases? What's the distance in such cases, i.e. how many operations does it take to transform one string into the other?
- What's the edit distance if both strings have the same head?
- If the heads are different, what are the three ways of transforming the left string into the right string? The edit distance will be the lowest of the corresponding edit distances.

Complete the following.

- if $\text{left} = \dots$: $\text{edit}(\text{left}, \text{right}) = \dots$
- if $\text{right} = \dots$: $\text{edit}(\text{left}, \text{right}) = \dots$
- if $\text{head}(\text{left}) = \text{head}(\text{right})$: $\text{edit}(\text{left}, \text{right}) = \dots$
- otherwise: $\text{edit}(\text{left}, \text{right}) = \text{lowest of}$
 - \dots
 - \dots
 - \dots

Hint Answer

Exercise 24.4.2

Implement the recursive definition, using `...[0]` and `...[1:]` for the head and tail operations.

The `timeit` line in this and the following code cells helps you see how the different approaches improve the run-time.

```
[2]: def edit(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""
    pass

test(edit, edit_tests)
%timeit edit('algorithm', 'logarithm')
```

Hint Answer

Exercise 24.4.3

Copy your code to the next cell and modify it so that it uses indices `l` and `r` instead of slicing the strings.

```
[3]: def edit_indices(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""

    def edit(l: int, r: int) -> int:
        """Return the Levenshtein distance of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        """

        pass

    pass # call the inner function and return the result

test(edit_indices, edit_tests)
%timeit edit_indices('algorithm', 'logarithm')
```

Hint Answer

Exercise 24.4.4

Explain why dynamic programming is applicable to the Levenshtein distance problem.

Hint Answer

Exercise 24.4.5

Compute the Levenshtein distance with a top-down dynamic programming algorithm. Copy your Exercise 24.4.3 code to the next cell and add a cache.

```
[4]: def edit_topdown(left: str, right: str) -> int:  
    """Return the Levenshtein distance between the strings."""  
  
    def edit(l: int, r: int) -> int:  
        """Return the Levenshtein distance of left[l:] and right[r:].  
  
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)  
        """  
        pass  
  
    pass  
  
test(edit_topdown, edit_tests)  
%timeit edit_topdown('algorithm', 'logarithm')
```

Hint Answer

Exercise 24.4.6

For bottom-up dynamic programming, in which order must the cache be filled?

Hint Answer

Exercise 24.4.7

Compute the Levenshtein distance with bottom-up dynamic programming. Copy your Exercise 24.4.5 code to the next cell and modify it.

```
[5]: def edit_bottomup(left: str, right: str) -> int:  
    """Return the Levenshtein distance between the strings."""  
    pass  
  
test(edit_bottomup, edit_tests)  
%timeit edit_bottomup('algorithm', 'logarithm')
```

Answer

Exercise 24.4.8

What's the worst-case complexity?

Hint Answer

24.5 Higher and higher

Given a grid of non-negative integers, what is the length of the longest path of ascending numbers, if we only move vertically or horizontally to an adjacent integer? Consider the following grid.

Figure 24.5.1

10	7	10
3	90	82

The longest ascending path is (7, 10, 82, 90) and therefore the answer is 4. Paths (3, 10) and (10, 82, 90) also ascend, but they're shorter. Path (3, 82, 90) isn't valid because 82 isn't adjacent to 3. Path (3, 7, 10, 82, 90) isn't valid because 3 and 7 are diagonally adjacent.

If there were any paths with duplicate integers, they wouldn't be valid either because the sequence must be strictly ascending.

```
[1]: from algoesup import check_tests, test

higher_tests = [
    # case           grid           length
    ('one longest path',   [[10, 7, 10], [3, 90, 82]], 4),
    ('all numbers equal',  [[2, 2], [2, 2]], 1),
    ('go around grid',    [[2, 3], [1, 4], [6, 5]], 6),
    ('two longest paths', [[6, 7, 8], [6, 5, 4], [3, 2, 1]], 5)
]

check_tests(higher_tests, [list, int])
OK: the test table passed the automatic checks.
```

24.5.1 Exercises

Exercise 24.5.1

Explain why this problem can be solved with backtracking.

Hint Answer

Exercise 24.5.2

Explain which *backtracking template* you will use and how you will adapt it to this problem, by answering these questions:

- Which of the templates applies to this problem?
 - all permutations satisfying the constraints
 - best permutation satisfying the constraints
 - all subsets satisfying the constraints
 - best subset satisfying the constraints?
- What are the candidates and the extensions?
- When is an extension compatible with the current candidate?
- What are the local and global constraints?
- When is a candidate a solution?
- What is the initial best solution?

Answer

Exercise 24.5.3

Implement your algorithm.

```
[2]: def higher(grid: list) -> int:
    """Return the length of the longest path of ascending numbers in
    →grid.

    Preconditions: grid is a table of non-negative integers
                   with r > 0 rows and c > 0 columns
    """
    pass

test(higher, higher_tests)
```

Hint Answer

Exercise 24.5.4

There's another algorithmic technique that solves this problem more efficiently than backtracking. Which technique is it and why is it more efficient? (You're not asked to outline an algorithm: you only have to justify that a more efficient alternative technique exists.)

Hint Answer

Exercise 24.5.5 (optional)

Implement the more efficient approach.

```
[3]: def higher(grid: list) -> int:  
    """Return the length of the longest path of ascending numbers in  
    →grid.  
  
    Preconditions: grid is a table of non-negative integers  
    with r > 0 rows and c > 0 columns  
    """  
    pass  
  
test(higher, higher_tests)
```

Hint

CHAPTER 25

TMA 03 PART 1

This study-free week is for you to catch up if you need to, and to complete the first part of TMA 03. If you haven't done so yet, download TMA 03 from the '[Assessment](#)' tab of the M269 website, and put it in a `TMA03` subfolder of your M269 folder.

TMA 03 draws on the whole module: it is more demanding and contributes to a larger percentage of your grade than previous TMAs. Before attempting TMA 03, I suggest you look again at your tutor's feedback on TMA 01 and TMA 02.

Since TMA 03 also covers material from Chapters 1–20, you can start working on Question 1 this week, even if you're behind schedule. Your tutor *cannot* give you an extension for TMA 03, so I advise to start well in advance of the deadline.

This chapter provides some advice that complements the summaries of Chapters [21 \(graphs\)](#), [22 \(backtracking\)](#) and [23 \(dynamic programming\)](#).

The guidance in Chapters [5](#), [10](#), [15](#) and [20](#) still applies.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

25.1 Problems on graphs

Graphs are incredibly versatile: they can represent any collection of entities and binary relations between them. Many problems can be solved by modelling the input with a graph and applying a graph algorithm.

There are two ways to approach such problems, and you may need to switch between both a few times to get a solution to your problem.

25.1.1 Starting with the graph

The first approach is to start by modelling the problem, i.e. thinking about the required graph:

- What are the entities, i.e. what do the nodes represent: people, places, states, tasks, events, ...?

- If nodes represent states, what are the possible states and how are they represented?
- What is the relation between entities, i.e. what do the edges represent? When is there an edge between nodes A and B?
- Is the relation symmetric or not, i.e. are the edges undirected or directed? If the latter, what do the directions represent?
- Are the edges weighted? Do weights represent distance, time or cost?

For problems on games, you probably need a state transition graph where the nodes represent the possible configurations of the game and the edges represent the possible moves.

The current configuration of a game is the result of all the previous moves. More generally, a problem that involves remembering what happened before most likely needs to be modelled with a state transition graph, where each state contains the necessary information to make the next transition. For example, in the *rook's moves* problem, every move depends on the previous one, so nodes must represent the state of the rook, not just its position. State transition graphs are usually directed and often weighted, where the weight typically represents the time or cost of the transition.

25.1.2 Starting with the general problem

An alternative approach is to think which graph problem you learned about is most closely related to the problem you have at hand and let that guide your modelling. The general problems you learned about are:

1. solving the TSP for complete weighted graphs with *exhaustive search* or *backtracking*
2. solving the single-source shortest paths (SSSP) problem for unweighted graphs, with *breadth-first search*
3. solving the SSSP for graphs with non-negative weights, with *Dijkstra's algorithm*
4. computing a *minimum spanning tree* of a weighted undirected graph, with Prim's algorithm
5. computing the components of a *directed* or *undirected* graph, with any form of graph traversal
6. computing a *topological sort* of a directed acyclic graph, with Kahn's algorithm
7. detecting a cycle in a directed graph, with Kahn's algorithm.

The first four problems were already mentioned before. To spot whether the last three fit your problem, ask yourself:

- Is it a problem about connectivity, i.e. which entity is connected to which?
- Does the problem involve grouping entities, like forming *islands from land squares*, to then find the number of groups or the largest group?

If the answer is yes to either question, then consider computing components. It's likely the graph you need isn't weighted or directed.

- Does the problem ask for nodes by dependency order, like when scheduling events or tasks? Does it ask if no such order is possible?

- Do you have to check if there's a path from a node back to itself?

If the answer is yes to any of these questions, then consider using Kahn's algorithm to find a topological sort or to detect a cycle. In both cases the graph must be directed.

Finally, always remember to consider:

- Can the standard algorithm be applied as-is or must it be adapted for this problem?

Even if some modification of a graph traversal or of Prim's, Dijkstra's or Kahn's algorithm may be needed, it's safer to start from a tried-and-tested graph algorithm than inventing a new one from scratch.

25.2 Backtracking

You can apply backtracking to solve a problem with these properties:

1. It's a constraint satisfaction or optimisation problem.
2. Each solution is a set or a sequence of items.
3. The candidates can be constructed incrementally, item by item.
4. Each solution must satisfy local constraints that can be checked as each item is added.

In addition, the way backtracking is presented in M269, you must know in advance the extensions, i.e. the items that can be added to a candidate. Moreover, if a solution is a sequence, items should be unique, but it may possible to work around this limitation. For example, we can solve the TSP with backtracking even though one node appears twice in the tour.

If candidates can't be generated incrementally or if there are no local constraints, use brute-force search instead.

The following subsections present the code templates for constraint satisfaction and optimisation problems on sequences and sets.

25.2.1 Constraints on sequences

If the problem asks for all sequences that satisfy some constraints, consider the following:

- What are the candidates? What do they represent?
- What are the items in the candidates? What do items represent?
- What data structure should be used for each item?
- Is the initial candidate the empty sequence?
- What is the initial set of extensions? How can it be generated?
- When is a candidate a solution, i.e. what are the constraints?
- Which constraints are local (can be checked on partial candidates) and which are global (must be checked on complete candidates)?
- Can partial candidates be solutions, like in *Trackword*?

Then copy the following code template and adapt it to your problem, according to your answers to the above questions.

```
def problem(instance: object) -> list:
    """Return all solutions for the problem instance, in the order_
→generated."""
    candidate = [...]           # initial candidate, usually []
    extensions = ...            # a set of items
    solutions = []
    extend(candidate, extensions, instance, solutions)
    return solutions

def extend(candidate: list, extensions: set, instance: object,_
→solutions: list) -> None:
    """Add to solutions all extensions of candidate that solve the_
→problem instance."""
    # print('Visiting node', candidate, extensions)
    # remove the next line if partial candidates can be solutions
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            solutions.append(candidate)
    for item in extensions:
        if can_extend(item, candidate, instance):
            extend(candidate + [item], extensions - {item}, instance,
→ solutions)

def satisfies_global(candidate: list, instance: object) -> bool:
    """Check if candidate satisfies the global constraints."""
    return ...

def can_extend(item: object, candidate: list, instance: object) ->_
→bool:
    """Check if item may extend candidate towards a solution."""
    return ...
```

In this and the following templates, you may uncomment the print statements to debug your code and check if the search space is being pruned as you'd expect. Once your code passes the tests, make the variable names and docstrings specific to the problem and remove unnecessary parameters.

25.2.2 Best sequence

If the problem asks for one sequence that maximises or minimises some value, consider in addition to the previous questions:

- Is it a minimisation or maximisation problem?
- What value is being minimised or maximised?
- Is there an easily generated ‘good’ solution, i.e. with a low (or high) value?

The following assumes the value is an integer, but an optimisation problem can be about any type of comparable values. If they aren't comparable, it's impossible to determine the best solution.

```

SOLUTION = 0
VALUE = 1

def problem(instance: object) -> list:
    """Return the best solution the problem instance and its value."""
    candidate = [...] # initial candidate, usually []
    extensions = ... # a set of items
    solution = ... # ideally with a value near lowest/highest
    best = [solution, value(solution, instance)]
    extend(candidate, extensions, instance, best)
    return best

# in the next line replace 'int' by the value's type
def value(candidate: list, instance: object) -> int:
    """Return the value of the candidate sequence for the problem
    instance."""
    return ...

def extend(candidate: list, extensions: set, instance: object, best:_
          ->list) -> None:
    """Update best if candidate is a better solution, then extend it.
    """
    # print('Visiting node', candidate, extensions)
    # remove the next line if partial candidates can be solutions
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            candidate_value = value(candidate, instance)
            # in the next line, use < for minimisation problems
            if candidate_value > best[VALUE]:
                # print('New best with value', candidate_value)
                best[SOLUTION] = candidate
                best[VALUE] = candidate_value
    for item in extensions:
        if can_extend(item, candidate, instance):
            extend(candidate + [item], extensions - {item}, instance,
                   best)

def satisfies_global(candidate: list, instance: object) -> bool:
    """Check if candidate satisfies the global constraints."""
    return ...

def can_extend(item: object, candidate: list, instance: object) ->_
              ->bool:

```

(continues on next page)

(continued from previous page)

```
"""Check if item may extend candidate towards a solution."""
return ...
```

If no initial solution can be easily constructed, use a pseudo-solution with an artificial positive or negative infinite value. The first solution found by backtracking will be the first best solution. The main function becomes:

```
import math

def problem(instance: object) -> list:
    """Return the best solution the problem instance and its value."""
    ↪
    candidate = [...]           # initial candidate, usually []
    extensions = ...            # a set of items
    best = [[], math.inf]        # use -math.inf for maximisation problems
    extend(candidate, extensions, instance, best)
    return best
```

To further prune the search space, consider:

- Does extending a candidate worsen its value, i.e. does extending increase the value for a minimisation problem or decrease the value for a maximisation problem?

For the TSP, if weights are positive then extending a candidate path increases its length, which worsens the candidate's value because we're looking for a shortest tour. For problems like this, use the next `can_extend` template and add the `best` parameter to the call in function `extend`.

```
def can_extend(item: object, candidate: list, instance: object, best:
    ↪ list) -> bool:
    """Check if item can extend candidate into a better solution_
    ↪than best."""
    # replace ... with a check for the local constraints
    # use < for a minimisation problem
    return ... and value(candidate + [item]) > best[VALUE]
```

25.2.3 Constraints on sets

If the problem asks for all sets that satisfy some constraints, use the next template. The questions to consider are the same as for constraint problems on sequences, except that now the candidates are sets and the extensions form a sequence.

```
def problem(instance: object) -> list:
    """Return all solutions for the problem instance, in the order_
    ↪generated."""
    candidate = {...}      # initial candidate, usually set()
    extensions = ...        # a list of items
    solutions = []
```

(continues on next page)

(continued from previous page)

```

extend(candidate, extensions, instance, solutions)
return solutions

def extend(candidate: set, extensions: list, instance: object,_
→solutions: list) -> None:
    """Add to solutions all extensions of candidate that solve the_
→problem instance."""
    # print('Visiting node', candidate, extensions)
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            solutions.append(candidate)
    else:
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, instance, solutions):      #_
→add item
            extend(candidate.union({item}), rest, instance,_
→solutions)
            extend(candidate, rest, instance, solutions)           #_
→skip item

def satisfies_global(candidate: set, instance: object) -> bool:
    """Check if candidate satisfies the global constraints."""
    return ...

def can_extend(item: object, candidate: set, instance: object) ->_
→bool:
    """Check if item may extend candidate towards a solution."""
    return ...

```

If partial candidates can be solutions, then change the `extend` function to:

```

# print('Visiting node', candidate, extensions)
if satisfies_global(candidate, instance):
    solutions.append(candidate)
if len(extensions) > 0:
    item = extensions[0]

```

25.2.4 Best set

If the problem asks for one set that maximises or minimises some value, use this template. Consider the same questions as for the best sequence.

```

SOLUTION = 0
VALUE = 1

```

(continues on next page)

(continued from previous page)

```

def problem(instance: object) -> list:
    """Return the best solution the problem instance and its value."""
    ↪
        candidate = {...}      # initial candidate, usually set()
        extensions = ...       # a list of items
        solution = ...         # ideally with a value near lowest/highest
        best = [solution, value(solution)]
        extend(candidate, extensions, instance, best)
        return best

def extend(candidate: set, extensions: list, instance: object, best:_
↪list) -> None:
    """Update best if candidate is a better solution, then try to_
↪extend it."""
    # print('Visiting node', candidate, extensions)
    if len(extensions) == 0:
        if satisfies_global(candidate, instance):
            candidate_value = value(candidate, instance)
            # use < for a minimisation problem
            if candidate_value > best[VALUE]:
                # print('New best with value', candidate_value)
                best[SOLUTION] = candidate
                best[VALUE] = candidate_value
    else:
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, instance, best):
            extend(candidate.union({item}), rest, instance, best)
        extend(candidate, rest, instance, best)

```

As shown earlier, if partial candidates can be solutions then change the `extend` function, and if the initial best solution can't be easily constructed, start with a pseudo-solution of positive or negative infinite value.

To further prune the search space, consider:

- Is it possible to order the extensions by ‘incompatibility’, i.e. so that if an item can’t extend a candidate, none of the following can?

For the knapsack problem we can order items by ascending weight: if one item doesn’t fit the knapsack, nor do the subsequent heavier items. For problems like this, sort the extensions in the main function and use this `extend` function.

```

def extend(candidate: set, extensions: list, instance: object, best:_
↪list) -> None:
    """Update best if candidate is a better solution, then try to_
↪extend it."""
    # print('Visiting node', candidate, extensions)

```

(continues on next page)

(continued from previous page)

```

if satisfies_global(candidate, instance):
    candidate_value = value(candidate, instance)
    # use < for a minimisation problem
    if candidate_value > best[VALUE]:
        # print('New best with value', candidate_value)
        best[SOLUTION] = candidate
        best[VALUE] = candidate_value
    if len(extensions) > 0:
        item = extensions[0]
        rest = extensions[1:]
        if can_extend(item, candidate, instance, best):
            extend(candidate.union({item}), rest, instance, best)
            extend(candidate, rest, instance, best)
    
```

25.3 Dynamic programming

Dynamic programming is often applied to optimisation problems on sequences, because sequences are usually easy to process item by item to build up the problem's solution from the subproblems' solutions.

To obtain a dynamic programming algorithm, follow these steps.

- 1. Define the problem recursively.**

This is the key step and usually the hardest one.

The recursive definition has the same usual form. First define the base cases, which include the smallest inputs but possibly other cases too. Then define the recurrence relations: how the solution for a problem instance is obtained from solutions for subproblems (smaller input instances). For problems on sequences, use the head, tail and concatenation operations.

For optimisation problems, the recurrence relation must involve taking the best (typically the maximum or minimum) of two or more subproblem solutions. In the *LCS problem*, if the two string heads differ, we take the longest of two subsequences, by skipping the left or the right head.

- 1. Implement the recursive definition.**

Implement the recursive definition, which is usually straightforward, using `...[0]` for the head of a sequence and `...[1:]` for the tail. If the problem turns out to have overlapping subproblems, then the recursive algorithm may be exponential in the worst case, so test your code on small problem instances.

Once you have a working implementation, modify it to use indices instead of slicing. This improves the run-time and helps introduce a cache later.

When using indices, a recursive definition of function `f` on sequence `items` like

- if `items` is empty: `f(items) = ...`
- otherwise: `f(items) = ... head(items) ... tail(items) ...`

becomes a definition for a function f with an additional integer parameter $index$:

- if $index = |items|$: $f(items, index) = \dots$
- otherwise: $f(items, index) = \dots items[index] \dots index + 1 \dots$

The following code template uses an auxiliary inner recursive function that has only parameter $index$ and accesses $items$ from the main function.

```
def f(items: list) -> ...:
    def auxiliary(index: int) -> ...:
        if index == len(items):
            return ...
        else:
            return ... # based on items[index] and index+1

    return auxiliary(0) # start at index 0
```

3. Are there overlapping subproblems?

The next step is to show the existence of repeatedly solved subproblems, to justify the need for dynamic programming. You have to think whether it's possible to arrive at the same subproblem by making different choices whilst processing the input. It may help to draw the tree of recursive calls for a small problem instance to see if repeated nodes appear.

For example, in the *LCS problem*, whether we first skip the left head and then the right head or vice versa, we obtain the same subproblem. In the knapsack problem, *if two or more items have the same weight*, choosing any of them and skipping the others will lead to the same subproblem: the same remaining capacity and the same remaining items to choose from.

4. Define a cache data structure.

The subproblems are the possible value combinations for the inputs. The cache is a map of subproblems to their solutions.

If the inputs are one, two, three or more natural numbers, the cache can be implemented as a one-, two-, three- or higher-dimensional array, indexed by the input numbers. For example, the cache for the *LCS problem* is a two-dimensional array (a matrix) indexed by the indices of the left and right input strings.

If the input values are hashable, like a tuple or a string, the cache can be implemented with a hash table (a Python dictionary). If the input values are comparable but not hashable (like Python lists), consider a binary search tree for logarithmic lookup. You may also consider converting to a hashable data type, like tuples. If there's no efficient representation for the cache you need, consider going back to step 1 to find an alternative problem with inputs that suit dynamic programming better.

5. Implement top-down dynamic programming.

The next step is to modify the code of step 2 with the cache defined in step 4. The required modifications are usually rote. The original function

```
def f(instance: ...) -> ...:
    if instance is base case:
        return ...
    else:
        return ...
```

becomes

```
def f(instance: ...) -> ...:

    def auxiliary(instance: ...) -> ...:
        if instance not in cache:
            if instance is base case:
                cache[instance] = ...
            else:
                cache[instance] = ...
        return cache[instance]

    cache = ... # initialise cache
    return auxiliary(instance)
```

assuming the cache is a dictionary or array. If the cache is a dictionary, start it empty; if it's an array, initialise it with an impossible value that can't be the solution to any subproblem.

6. Implement bottom-up dynamic programming.

To develop the bottom-up approach, look at the recurrence relations or the code of the top-down approach to understand in which order to fill the cache. It may help to draw a DAG of the subproblems and their dependencies for a particular problem instance.

The DAG may be easiest to draw by thinking backwards, basically following the recursive call tree and merging common subproblems into one node. However, you may also draw the DAG starting with the base cases and proceeding in a breadth-first way: Which subproblems depend only on the base cases? Which subproblems then depend on the ones just added? In other words, think which subproblems ‘feed’ into which other ones.

The subproblems can then be solved in any topological order of the DAG. If the cache is a multi-dimensional array, it may be as simple to fill as using nested loops, one per dimension. Each loop goes through the corresponding indices in ascending (first to last) or descending (last to first) order. The code becomes:

```
def f(instance: ...) -> ...:
    cache = ... # initialise cache as in top-down approach

    for row in range(...):
        for column in range(...):
            if the instance (row, column) is a base case:
                cache[row][column] = ...
            else:
```

(continues on next page)

(continued from previous page)

```
cache[row][column] = ...  
  
return cache[...][...] # usually one of the corner cells
```

After filling the cache, the algorithm returns the entry corresponding to the input instance.

For example, in the *LCS problem* each matrix cell depends on cells below or to the right in the same row, so both the row and column loops must iterate backwards, in descending index order, and the solution for the input instance is in the top left-hand cell.

However, for the knapsack problem, each cell only depends on *cells below and to the left*, so we iterate through the rows in descending order and through the columns in ascending order. The final solution is in the top right-hand cell.

7. Analyse the complexity and measure the performance.

The worst-case complexity of dynamic programming (either approach) is the size of the cache multiplied by the worst-case complexity of computing each entry. The size of the cache is the number of subproblems, which is at most the product of all possible values for each input.

The top-down approach fills only the part of the cache that is needed for the input problem instance, while bottom-up fills the whole cache. On the other hand, the top-down algorithm makes recursive calls, while bottom-up is purely iterative. Even though the worst-case complexity is the same, the bottom-up approach may have lower run-times if the top-down approach fills most of the cache. You will have to measure the run-times of both approaches for typical inputs to assess which approach is best in practice.

CHAPTER 26

COMPLEXITY CLASSES

Biologists have classified plants and animals into a ranked taxonomy, formed of species, genus, family, order, and so on. This classification makes it easier to identify new species, relate them to existing ones and understand the evolution of organisms.

Likewise, computer scientists have classified problems to understand their similarities and differences. Computational problems are classified according to how efficiently they can be solved. For example, problems like *interval scheduling* and *topological sort* are in the same class, because both can be efficiently solved in less than quadratic time.

As you shall see later in this chapter, it's unknown whether problems like the travelling salesman problem (TSP) are in that same class. Has nobody been clever enough to find an efficient algorithm for the TSP, or is the TSP genuinely a harder problem than others? Computer scientists have put problems like the TSP in their own class. Knowing that a problem is in the same class as the TSP means that it's unlikely there's an efficient algorithm for that problem. That's useful knowledge. Instead of trying (and likely failing) to design an efficient correct algorithm, we should look for a different approach, e.g. an efficient approximate algorithm that returns a 'good enough' solution.

This chapter introduces six classes (sets) of problems and a versatile technique, called problem reduction, that helps classify problems but can also be used as an algorithmic technique.

This chapter supports these learning outcomes:

- Understand the common general-purpose data structures, algorithmic techniques and complexity classes – this chapter introduces problem reduction (as an algorithmic technique) and six complexity classes: tractable, intractable, P, NP, NP-hard, NP-complete.
- Analyse the complexity of algorithms to support software design choices – this chapter introduces problem reduction as a complexity analysis technique.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

26.1 Tractable and intractable problems

Before we look at two major classes of problems, we must transfer the notion of complexity from algorithms to problems.

26.1.1 Problem complexity

The classification of problems is based on how efficiently they can be solved, so the **complexity of a problem** is defined as the complexity of the most efficient algorithm that solves it. Like for algorithms, we could distinguish between the best-, average- and worst-case complexity of a problem, but for the classification of problems we're only interested in worst-case scenarios.



Note: In this chapter, we only consider worst-case complexities and I will therefore often omit the ‘worst-case’ adjective.

It's important to realise that the complexity of a problem is the lowest complexity of *all* algorithms that solve the problem, including those that haven't been discovered yet. To be able to say that a problem has, say, quadratic complexity, we must construct a quadratic algorithm that solves the problem *and* we must *prove* that it's impossible to write a more efficient algorithm.

Consider the problem of sorting comparable items. This problem has log-linear complexity because there are log-linear algorithms that solve it, like merge sort and heapsort, and there's *a proof* that a log-linear number of comparisons is needed to sort the items. Hence no algorithm with a lower complexity is possible. (More efficient algorithms, like pigeonhole sort, only work for items that can be sorted without comparing them.)

For many problems, like the TSP, their exact complexity is unknown because there's currently no proof that the most efficient algorithm we know of is also the most efficient algorithm there will ever be. In other words, there's no proof that a more efficient algorithm is impossible.

All we can say for such problems is that their complexity is *at most* the complexity of the most efficient known algorithm and *at least* the complexity it takes to produce the output. For example, the complexity of the TSP is at least linear, because it takes linear time to copy all nodes to the output tour, and at most exponential, because the best known algorithm has that complexity. The exact complexity of the TSP could be linear, exponential, or anything in between. We don't know which one it is yet.

26.1.2 Tractable problems

A polynomial is an expression of the form $a_0 + a_1 \times n + a_2 \times n^2 + \dots + a_c \times n^c$, where the a 's and the c are constants and n is a numeric variable. For example, $5 + 3n + 2.5n^2 + 0.5n^4$ is a polynomial with $a_3 = 0$.



Info: MST124 Unit 3 Section 1.6 introduces polynomials.

In complexity analysis, only the highest term counts and constant factors are ignored, so

$$O(a_o + a_1 \times n + a_2 \times n^2 + \dots + a_c \times n^c) = O(n^c).$$

That's why we say that an algorithm with complexity $O(n^c)$ has **polynomial complexity**, or vice versa, that a **polynomial algorithm** has complexity $O(n^c)$.

Big-Oh indicates an upper bound, so having polynomial complexity means to have complexity $\Theta(n^c)$ or better. Therefore, any algorithm with constant, logarithmic, linear, log-linear, quadratic or cubic complexity is polynomial because it has complexity $\Theta(n^3)$ or better. Most M269 algorithms have polynomial complexity.

A problem is **tractable** if it has polynomial complexity, i.e. if the most efficient algorithm that solves the problem has complexity $\Theta(n^c)$ or better, for input size n and some constant c . Most M269 problems are tractable.

We don't always need to know the exact complexity of a problem to know if it's tractable: a single algorithm of polynomial complexity is sufficient. For example, if the only known sorting algorithm were insertion sort, that would be enough to show that the sorting problem is tractable, as follows:

1. Since the most efficient sorting algorithm can't be worse than insertion sort, it must have complexity $\Theta(n^2)$ or better.
2. Since the most efficient sorting algorithm has complexity $\Theta(n^c)$ or better for $c = 2$, it has by definition polynomial complexity.
3. Since the most efficient sorting algorithm has polynomial complexity, by definition the sorting problem has polynomial complexity.
4. Since the sorting problem has polynomial complexity, it is by definition tractable.

Usually, we don't spell out all these steps: we simply say that the sorting problem is tractable because it is solved by a polynomial algorithm, e.g. insertion sort, which has quadratic complexity.



Note: To show that a problem is tractable you only need to indicate or construct *one* polynomial algorithm that solves the problem.

26.1.3 Intractable problems

A problem is **intractable** if it's not tractable: there's no polynomial algorithm for it. The most efficient algorithm for an intractable problem has complexity higher than $O(n^c)$. There are many such complexities but in M269 you learned only two: the exponential and factorial complexities $\Theta(2^n)$ and $\Theta(n!)$.

A problem that can be solved with an exponential or factorial algorithm isn't necessarily intractable: there might be a polynomial algorithm that solves it, making the problem tractable. For example, the sorting problem can be solved with the factorial *bogosort* algorithm, but the sorting problem is actually tractable, as explained above.

To classify a problem as intractable we must prove that no polynomial algorithm solves it. There's a case where this can be easily proven, namely if the size of the output is exponential or factorial in the size of the input. In that case, even if each item in the output can be computed in constant time, it will take an exponential or factorial time to produce the output, and therefore no polynomial algorithm for it can exist.

For example, the problem of *computing all subsets* of a given set with n items is intractable, because there are 2^n subsets. Even if each subset were produced in constant time, it would take exponential time to produce all of them, and so no polynomial algorithm can solve the problem.



Note: If the size of the output is exponential or factorial in the size of the input, then the problem is intractable.

In Python, floating-point numbers have a fixed size (64 bits), but integers can be arbitrarily large. So be careful when analysing the complexity of problems that have integer inputs, because the complexity is the growth rate of the run-time with respect to the input *size*, not the input *value*.

For example, the multiplication of integers x and y takes constant time for 64-bit integers. But for arbitrarily large integers, an algorithm that multiplies each digit of x with each digit of y has complexity $\Theta(|x| \times |y|)$, the product of the sizes of both numbers.

The size of an integer n is $|n| = \log_2 n$ because that's the least number of bits required to store n . For example, $|23| = \log_2 23 = 4.52$. In fact, 23 is written 10111 in binary, which takes 5 bits, but we'll ignore the rounding up because it doesn't affect the complexity.

Since the logarithm and exponentiation are inverse operations, we have $n = 2^{\log_2 n} = 2^{|n|}$. So, if we have a complexity expression in terms of the value n , then the expression in terms of the size $|n|$ is obtained by replacing n with $2^{|n|}$, which in turn can be replaced with $2^{\log n}$ because we *ignore the base of logarithms* when analysing complexity.

For example, our *factorisation algorithm* for a positive integer n has complexity $\Theta(\sqrt{n}) = \Theta(n^{0.5})$. This means the factorisation algorithm is polynomial in the input *value* because the complexity is of the form $\Theta(n^c)$. However, to express the complexity in terms of the input *size* we must write

$$\Theta(n^{0.5}) = \Theta((2^{|n|})^{0.5}) = \Theta((2^{\log n})^{0.5}) = \Theta(2^{0.5 \log n}) = \Theta(2^{\log n}).$$

(Remember that complexity analysis ignores constant factors.) It thus turns out that factorisation is exponential in terms of the input size, which is the $\log n$ bits required to represent n .



Info: MU123 Unit 3 Section 1.5 explains why $(x^y)^z = x^{y \times z}$ and Section 3.4 explains why $\sqrt{x} = x^{0.5}$.

Algorithms that have polynomial complexity in the *value* (but not in the *size*) of an integer input are called **pseudo-polynomial**.



Note: A pseudo-polynomial algorithm seems polynomial but is in fact exponential.

26.1.4 The twilight zone

For some problems, we don't know if they're tractable or not. This happens when the currently most efficient algorithm is *not* polynomial but the output's size *is* polynomial in the size of the input and so a polynomial algorithm *might* be possible.

For example, I wrote earlier that the exact complexity of the TSP is at least linear and at most exponential. So the problem could be tractable or intractable.

- If someone invents a polynomial algorithm for the TSP, then we know it's tractable.
- If someone proves that there can't be a polynomial algorithm for the TSP, then we know it's intractable.



Note: If a problem's complexity is at most non-polynomial but could be polynomial, then we don't know if the problem is tractable or intractable.

Exercise 26.1.1

Based only on what you have read in this book, is the *interval scheduling* problem tractable, intractable or can't you say either way?

Hint Answer

Exercise 26.1.2

Based only on what you have read in this book, is the *0/1 knapsack* problem tractable, intractable or can't you say either way?

Hint Answer

Exercise 26.1.3

A group of tourists is visiting a city on foot. They start the day in their hotel. They know the restaurants where they will have lunch and dinner. They will visit a museum immediately after lunch.

Given an unweighted, undirected, connected graph representing the city's places and streets, we want to compute all paths from node *Hotel* to node *Dinner* that go through nodes *Lunch* and *Museum* one after the other.

Is this problem tractable, intractable or can't you say?

Hint Answer

Exercise 26.1.4

Are pseudo-polynomial algorithms intractable? (This may be considered a trick question.)

Hint Answer

26.2 The P and NP classes

This section is about decision problems, i.e. those with a Boolean output. There are two major classes of decision problems, called P and NP. I will use the arguably most famous decision problem to introduce both classes.

26.2.1 SAT

Consider the Boolean expression ‘not (A or (B and C))’. Depending on the values of the three variables, the expression may be true or false. If A is true, then the expression is false, but if A and B are false, then the expression is true.

An **interpretation** is a set of assignments of a Boolean value to each variable. For example, the interpretations

- {A = true, B = false, C = false}
- {A = true, B = true, C = true}

make the expression false, while the interpretations

- {A = false, B = false, C = false}
- {A = false, B = false, C = true}

make the expression true. A Boolean expression with n variables has 2^n different interpretations, because each variable has two possible values.

A Boolean expression is **satisfiable** if at least one interpretation satisfies the expression, i.e. makes it true. The expression ‘not (A or (B and C))’ is satisfiable, but ‘A and not A’ isn’t: neither {A = false} nor {A = true} makes the expression true. The **satisfaction problem** (also known as **SAT**) is the problem of deciding whether a given Boolean expression is satisfiable.

SAT can be solved with brute-force search: generate all possible interpretations for the input expression and evaluate the expression with each interpretation. If a candidate interpretation makes the expression true, stop searching and output true: the expression is satisfiable. Otherwise, output false after generating and testing all interpretations.

Testing one interpretation takes linear time: one pass over the expression to replace the variables with their values (as given by the interpretation) and one pass to evaluate the Boolean operators, each in constant time.

In the worst case, the search generates and tests all 2^n interpretations to realise that none of them (or only the last one) makes the expression true. There are SAT algorithms that are quite efficient in the average case, i.e. for most Boolean expressions, but all known SAT algorithms have exponential worst-case complexity.

26.2.2 Class P

Class P is the set of tractable decision problems: those that can be solved in polynomial time. For example, the problem of deciding whether a given string is a *valid password* is in P, because a simple linear-time algorithm solves it.

As I mentioned before, for some problems like the TSP we don't know if they're tractable, and the same happens with some decision problems. For example, we don't know whether SAT is tractable, i.e. if it's in P or not, because while current SAT algorithms are exponential, there's no proof that a polynomial algorithm for SAT can't exist.

26.2.3 Class NP

If for some Boolean expression the output for the SAT problem is true, then we can verify in linear time that the expression is indeed satisfiable, provided we're given an interpretation that satisfies the expression: we simply evaluate the expression with the interpretation to confirm it makes the expression true.

In more general terms, SAT is a decision problem with this property: for every input that leads to a true output (the decision is 'yes'), we can provide some data that allows us to confirm the decision in polynomial time. **Class NP** is the set of all decision problems with this property.

Class P is the set of decision problems for which the 'yes' or 'no' decision can be *computed* in polynomial time; class NP is the set of decision problems for which a 'yes' decision can be *checked* in polynomial time.

The additional data to check the 'yes' decision is called a **certificate** because it certifies that the output must be true for that input. The polynomial algorithm that takes an input and its associated certificate to confirm the decision must be 'yes' is called the **verifier**. For each input for which the decision is 'yes', the certificate is the extra information needed to verify the decision.

To show that a decision problem is in NP, we must

1. define a certificate for each input that leads to a 'yes' decision
2. outline the verifier's algorithm, explaining why it does confirm 'yes' decisions
3. justify that the algorithm has polynomial complexity.

Here's how I would answer a TMA question asking to show that SAT is in NP:

1. If the output is true, the input is a satisfiable expression. The associated certificate is an interpretation that satisfies the expression.
2. The verifier takes an input expression and an interpretation, and evaluates the expression using the interpretation. If the expression evaluates to true, this confirms the expression is satisfiable because this interpretation makes the expression true.
3. If the expression has v variable occurrences and o Boolean operators, the verifier takes linear time in the size of the expression: $\Theta(v + o)$. It takes $\Theta(v)$ to replace the variables with the interpretation's values and $o \times \Theta(1)$ to evaluate the operators.

As a further example, consider the decision problem of whether a sequence has even length. It can be shown to be in NP:

1. The certificate is an integer: the length of the input sequence.
2. The verifier takes the list and its certificate and checks that the certificate is an even number that corresponds to the length of the sequence. This confirms that the sequence has even length.
3. The verifier takes constant time to check that the certificate is an even number, using the modulo operation. The verifier takes at most linear time to compute the length of the sequence and check it's equal to the certificate.

Note that only checking if the certificate is an even number isn't enough to confirm the list has even length: it could be that the certificate was incorrectly computed. The verifier is checking that a given input leads to a 'yes' decision, so the verifier can't use only the certificate and ignore the input.

For this example we can produce certificates for all input sequences, not just those of even length, but in general you only have to indicate what the certificate is for each input leading to a true output.

Exercise 26.2.1

Alice has an extended lunch break during a conference. Can she walk around the city to see all the major landmarks and return to the hotel within 2 hours? Her problem is a particular case of the **decision TSP**:

Given a complete weighted graph and a positive integer w , does the graph have a tour with total weight less than or equal to w ?

1. What is the certificate for a graph and integer that lead to a 'yes' answer?
2. Outline the verifier algorithm.
3. Explain why the verifier takes polynomial time.

Hint Answer

26.2.4 P versus NP

When I proved that deciding whether a sequence has even length is an NP problem, you may have noticed that the certificate is redundant because the verifier computes the length of the sequence anyway. The verifier is in effect the decision algorithm: it computes the length and checks it is even.

Deciding if a sequence has even length is a tractable problem and so we can use its polynomial algorithm to confirm a 'yes' decision without really making use of the certificate.

The argument that every tractable decision problem is in NP goes as follows:

1. For every input, the certificate can be anything: zero, the empty set, the string 'whatever', etc.
2. The verifier algorithm takes an input and its certificate, ignores the certificate, calls the decision algorithm on the input and checks the output is true, to confirm it.

3. The decision algorithm (and therefore the verifier) takes polynomial time because the problem is tractable.

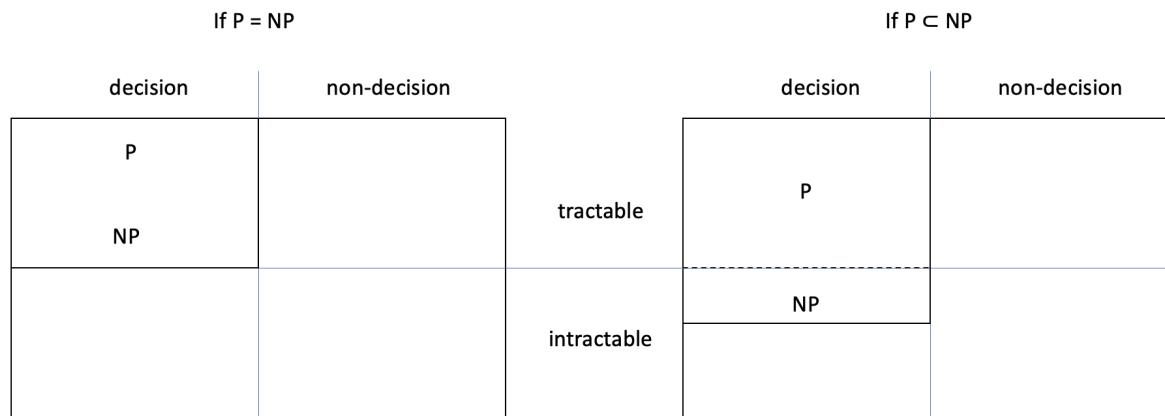
In summary, if a decision can be computed in polynomial time, then it can be verified in polynomial time. Therefore, every decision problem that is in set P is also in set NP.



Note: $P \subseteq NP$.

If P is a subset of NP, do we have $P = NP$ or $P \subset NP$ (and thus $P \neq NP$)? The next figure shows both possibilities. I've added empty columns for non-decision problems to emphasise that classes P and NP are only about decision problems.

Figure 26.2.1



In the left-hand diagram, P and NP are the same set: the tractable decision problems. In the right-hand diagram, $P \subset NP$: NP includes P (as conveyed by the dashed line) and includes the intractable decision problems that can be verified in polynomial time. There are further intractable decision problems, outside class NP.

Which of these two possibilities is actually the case? Is every NP problem in P (and therefore $P = NP$) or are some NP problems not in P (and therefore $P \neq NP$)? Asking if $NP = P$ can also be phrased as: is every NP problem tractable? In other words, for *every* decision problem for which we can *check* the ‘yes’ decisions in polynomial time, can we also *compute* them in polynomial time?

This is known as the **P versus NP problem** or the **‘P = NP?’ question**. It is literally a million-dollar question. Since 2000, the Clay Mathematics Institute has been offering one million dollars for a proof of $P = NP$ or of $P \neq NP$.



Info: You can find informal and formal descriptions of the problem, together with the rules for claiming the prize, on [their website](#).

Here are two suggestions for how you can get a million bucks. (No need to thank me.)

To prove that $P \neq NP$, you ‘just’ have to prove that *one* NP problem of your choice is intractable. Since all problems in P are tractable by definition, this would show that P and NP are not the same set of problems.

To prove $P = NP$, you ‘just’ have to invent a polynomial algorithm for SAT. I know, it sounds unbelievable that inventing an efficient algorithm for *one* particular problem proves that *all* the infinitely many NP problems can also be solved (not just verified) efficiently. As Tolkien might have noted if he had known about SAT: one problem to bind them all.

The next two sections will explain why a polynomial algorithm for SAT ‘unlocks’ polynomial algorithms for all NP problems. That’s why the P v. NP issue is so famous and significant, and why there’s a large bounty on an elusive algorithm or proof that settles the issue.

The majority of computer scientists believe that $P \neq NP$, partly because no polynomial algorithm for SAT has been found in the past 50 years, since the study of the P v. NP question began. The key word in the previous sentence is ‘believe’, because there’s no certainty either way.

26.3 Reductions

This section introduces problem reduction as an algorithmic technique. (The next section will show how to use problem reduction to classify problems.)

Imagine I’m asked to write an algorithm for problem *Unsolved*. It just happens that I know of a similar problem *Solved* for which there is an algorithm. I can then write the following algorithm to handle *Unsolved*:

1. Transform the inputs of *Unsolved* into the inputs of *Solved*.
2. Apply the known algorithm to obtain the output of *Solved*.
3. Transform the output of *Solved* into the output of *Unsolved*.

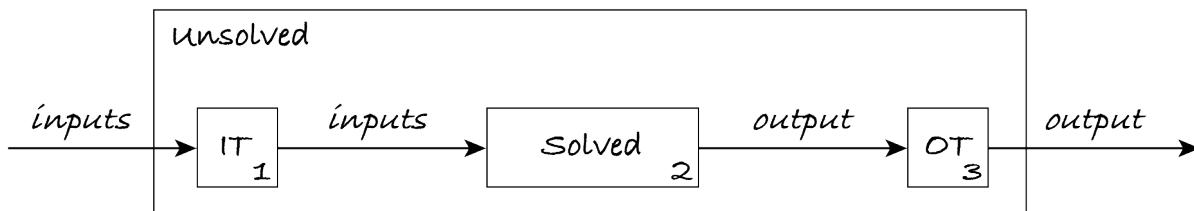
This algorithm is called a **reduction** of problem *Unsolved* to problem *Solved*.



Info: There are several kinds of reductions. This one is called a mapping reduction because it maps the inputs and output of one problem to those of another problem.

The next figure shows the reduction schematically. Each box is an algorithm, where IT and OT stand for input and output transformation. The numbers in the boxes are the corresponding steps above.

Figure 26.3.1



The complexity of the algorithm for *Unsolved* is the sum of the complexities of the two transformations and of the algorithm for *Solved*. If transforming the inputs and output takes longer than the algorithm for *Solved*, then reducing *Unsolved* to *Solved* is not very helpful. However, if we find a problem *Solved* that makes the transformations easy and efficient, then reduction might be a good approach to quickly write an initial algorithm for *Unsolved*.



Note: Normally we're only interested in reducing problem *Unsolved* to problem *Solved* if the complexity of the transformations is lower than the complexity of *Solved*.

Let's look at some examples.

26.3.1 Median

Let's assume that *Unsolved* is the problem of finding the median of a sequence of numbers. Since the median is the middle number (or the mean of two middle numbers) if they're sorted, we can reduce the median problem to the sorting problem.

The following code implements the reduction algorithm. I use generic variable names and some unnecessary assignments just to make the various steps of the reduction clearer.

```
def median(numbers: list) -> float:
    """Return the median of the numbers.

    Preconditions: numbers isn't empty
    """
    # finding the median is the unsolved problem
    unsolved_input = numbers

    # transform input of solved problem into input of unsolved_
    ↪problem
    solved_input = unsolved_input

    # obtain output of solved problem by applying a known algorithm
    solved_output = sorted(solved_input)

    # transform output of solved problem into output of unsolved_
    ↪problem
    length = len(solved_output)
    middle = length // 2
    # if the length is odd, the median is the middle number
    if length % 2 == 1:
        unsolved_output = solved_output[middle]
    # otherwise it's the mean of the two middle numbers
    else:
        left = solved_output[middle]
```

(continues on next page)

(continued from previous page)

```

        right = solved_output[middle + 1]
        unsolved_output = (left + right) / 2
    return unsolved_output
    
```

The input transformation just passes the list of numbers to the sorting algorithm, in constant time. The output transformation is not as simple, as it has to distinguish odd- and even-length sequences, but it also takes constant time. Since both transformations take constant time, the complexity of this algorithm is the complexity of sorting: the median problem can be solved in log-linear time by reducing it to the sorting problem.

26.3.2 Minimum and maximum

Consider the problem of finding the smallest value of a non-empty sequence of comparable values. It's similar to (it's actually a special case of) the *selection problem*: return the n -th smallest value of a sequence with n or more values. There's an efficient algorithm for this problem (quickselect), so we can reduce the problem of finding the smallest value to the problem of finding the n -th smallest.

```

def minimum(items: list) -> object:
    """Return the smallest item in items.

    Preconditions:
    - all items are pairwise comparable
    - len(items) > 0
    """
    # transform minimum problem input to selection problem input
    solved_items = items
    solved_n = 1
    # solve the selection problem
    solved_output = quick_select(solved_items, solved_n)
    # transform selection problem output to minimum problem output
    unsolved_output = solved_output
    return unsolved_output
    
```

The input transformation adds the second parameter that the selection problem requires: finding the minimum is finding the first smallest value, so $n = 1$. The output transformation just passes on the output of the selection problem.

The transformations take constant time and quickselect takes linear time, so the minimum can be found in linear time. Even though the reduction has the same complexity as a linear search for the minimum, the latter is a simpler and faster algorithm than quickselect.



Note: A reduction is often not the simplest algorithm for a problem.

Exercise 26.3.1

Reduce the problem of finding the maximum value in a non-empty sequence of comparable values to the selection problem. You only need to indicate how you would change the code above.

Hint Answer

26.3.3 Interval scheduling

To show that one problem reduces to another, we can write code as above, or just outline the input and output transformation algorithms. Here's an example, using the *interval scheduling* and *maximal independent set* problems.

Operation: largest interval schedule

Inputs: *intervals*, a set of pairs of natural numbers

Preconditions: for every pair $(start, end)$, $start < end$

Output: *schedule*, a set of pairs of natural numbers

Postconditions: *schedule* is a largest subset of *intervals* such that for any $(s_1, e_1) \neq (s_2, e_2)$ in *schedule*, either $e_1 < s_2$ or $e_2 < s_1$

Operation: maximal independent set

Inputs: *items*, a set of objects; *incompatible*, a set of pairs of objects

Preconditions: every pair in *incompatible* consists of two different members of *items*

Output: *compatible*, a set of objects

Postconditions: *compatible* is a largest subset of *items* that has no pair of elements from *incompatible*

Previously, we reduced the specific problem of finding the minimum (or maximum) to the general problem of finding the n -th smallest value. The general problem had one extra input, n .

Similarly, the interval scheduling problem is a specific version of the independent set problem: we want a largest subset of intervals that aren't incompatible (overlapping). The incompatibility relation is implicitly given by the start and end times of the intervals, whereas the generic problem has an extra input indicating which elements are incompatible.

If there's an algorithm for a general problem we can usually reuse it to solve more specific problems.

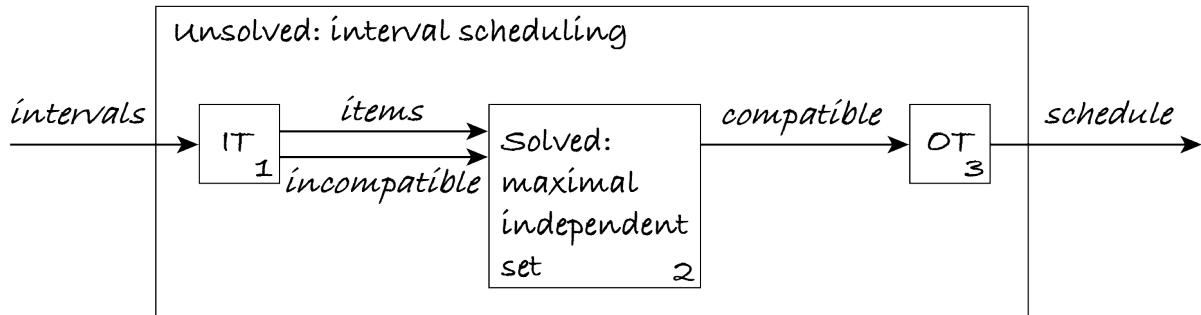


Note: We can usually reduce a specific problem to a more general one.

Reduction

The next figure shows the reduction of the interval scheduling problem (*Unsolved*) to the maximal independent set problem (*Solved*), with their inputs and outputs.

Figure 26.3.2



The input transformation algorithm constructs the inputs of the *Solved* problem (*items* and *incompatible*) from the inputs of the *Unsolved* problem (*intervals*) as follows:

1. let *items* be *intervals*
2. let *incompatible* be the empty set
3. for every *interval 1* in *intervals*:
 1. for every *interval 2* in *intervals*:
 1. if *interval 1* ≠ *interval 2* and not compatible(*interval 1*, *interval 2*):
 1. add (*interval 1*, *interval 2*) to *incompatible*

The auxiliary function ‘compatible’ checks that one interval ends before the other one starts.

Now the output transformation algorithm: it constructs the output of *Unsolved* (*schedule*) from the output of *Solved* (*compatible*).

1. let *schedule* be *compatible*

The input transformation constructs the set of overlapping pairs of intervals. The output transformation simply passes on the result: any algorithm that computes a maximal independent set also solves the interval scheduling problem, i.e. computes a largest subset of non-overlapping intervals, because the input *items* are the intervals and the *incompatible* pairs are the overlapping intervals.

Complexity

To determine the complexity of a reduction, we must add the complexities of the transformations and of the most efficient algorithm for problem *Solved*.

What are the complexities of the input transformation and of the output transformation for the above reduction?

The output transformation takes constant time. The input transformation generates all pairs of intervals in quadratic time and checks each one in constant time by just comparing the integer

start and end times. So the input transformation has quadratic complexity in the number of intervals.

The most efficient algorithm we know for the maximal independent set is *brute-force search*: go through the 2^n subsets of the n items, from largest to smallest subset, and stop when a compatible subset is found.

This means that problem reduction leads to an interval scheduling algorithm with exponential worst-case complexity. This is much worse than the log-linear greedy algorithm for interval scheduling.



Note: Reduction may be a way of quickly obtaining an initial algorithm for a problem, but it may be a very inefficient algorithm.

26.3.4 Reduction template

If you are asked in a TMA to reduce a given *Unsolved* problem to a *Solved* problem of your choice, you have to indicate the name of your chosen problem, its inputs and output, and the input and output transformations. You can present the transformations as step-by-step algorithms or just outline them. We suggest you present a reduction as follows.

	Unsolved	Solved
Prob-lem:	interval scheduling	maximal independent set
Input:	<i>intervals</i> , a set of integer pairs	<i>items</i> , a set of objects; <i>incompatible</i> , a set of pairs of objects
Output:	<i>schedule</i> , a set of integer pairs	<i>compatible</i> , a set of objects

Input transformation (*Unsolved* to *Solved*): Let *items* be *intervals*. Let *incompatible* be the empty set. For each pair of intervals, if one starts before the other one ends, then add the pair to *incompatible*.

Output transformation (*Solved* to *Unsolved*): Let *schedule* be *compatible*.

Exercise 26.3.2

Complete the template below to show how the problem of finding the maximum can be reduced to the sorting problem.

Problem:	maximum
Input:	<i>items</i> , a sequence of objects
Output:	<i>largest</i> , an object

Input transformation (*Unsolved* to *Solved*):

Output transformation (*Solved* to *Unsolved*):

What is the complexity of your reduction?

Hint Answer

Exercise 26.3.3 (optional)

Describe the reductions of Sections 26.3.1 and 26.3.2 using the template.

26.4 Problem hardness

So far we used reduction to obtain an algorithm for an unsolved problem, given an algorithm for an already-solved problem. This section shows how reduction can be used to classify problems.

26.4.1 Comparing problems

By reducing problem A to problem B, we obtain an algorithm for A: it transforms the inputs, uses the most efficient algorithm for B and transforms the output. So, the complexity of problem A is at most the sum of the complexities of the input and output transformations and of problem B. (Remember that the complexity of a problem is the complexity of the most efficient algorithm that solves it.)

We say ‘at most’ because there could be a more efficient algorithm for A, possibly without using reduction. For example, the reduction of the interval scheduling to the maximal independent set problem shows that the complexity of the former is at most exponential, and in fact there’s a far more efficient greedy algorithm.

We say that problem A is **at most as hard as** problem B, and B is **at least as hard as** A, if A can be reduced to B with two polynomial transformation algorithms.

All examples in the previous section use polynomial-time transformations, so finding the median is at most as hard as sorting, finding the minimum or maximum is at most as hard as selecting the n -th smallest value, and the interval scheduling problem is at most as hard as the maximal independent set problem.

Intuitively, A being at most as hard as B means that

- either A is as hard as B (both are tractable or both are intractable)
- or A is less hard than B (A is tractable and B is intractable).

This only happens if both transformations take polynomial time. If at least one transformation takes longer than polynomial time, then even if B is tractable, the complexity of the reduction algorithm for A is non-polynomial and A could be intractable (if no algorithm that is more efficient exists).

To sum up, if one or both transformations are non-polynomial it’s possible for A to be harder than B (A is intractable and B is tractable). With polynomial transformations, A is always at most as hard as B.

If A reduces in polynomial time to B and B is tractable, then so is A, because A can't be harder than B. More precisely, A is tractable because the sum of the polynomial complexities of the transformations and of the algorithm for B is polynomial. This is our first example of how reduction helps classify problems.



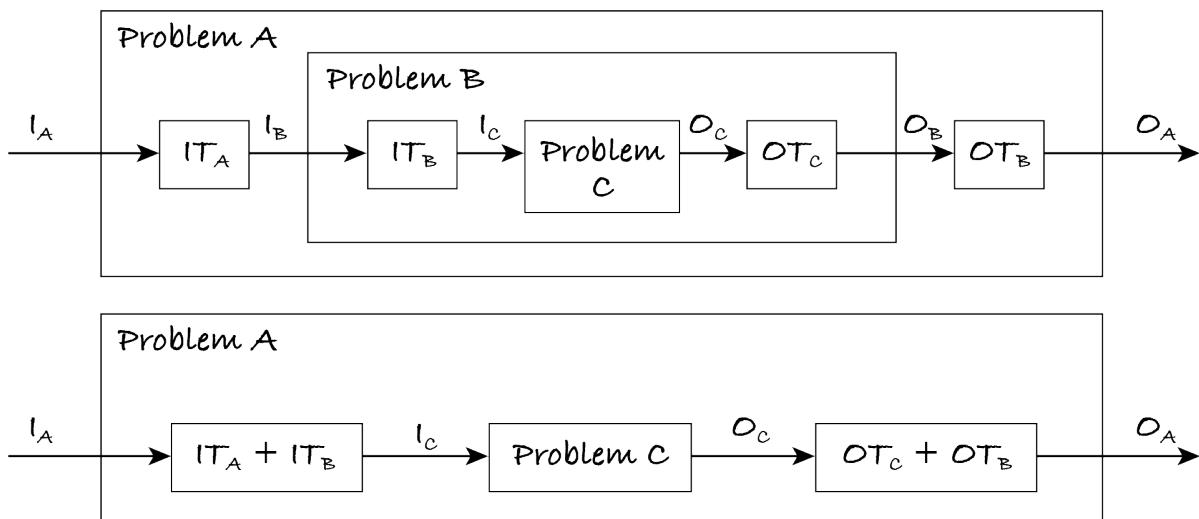
Note: If problem A reduces in polynomial time to a tractable problem, then A is tractable too.

26.4.2 Transitivity

Before I introduce the final two complexity classes, we must realise that reduction is *transitive*: if problem A reduces to problem B, and B reduces to problem C, then A reduces to C. The reason is simple: if we can transform A's inputs into B's and B's inputs into C's, then we can just write one transformation algorithm after the other to transform A's inputs directly into C's, and similarly for the output transformations.

The next figure shows that. The top diagram shows A being reduced to B, which is reduced to C, while the bottom diagram shows the direct reduction of A to C. Algorithms $IT_A + IT_B$ and $OT_C + OT_B$ are the concatenation of the input and of the output transformation algorithms.

Figure 26.4.1



Note: If A reduces to B and B reduces to C, then A reduces to C. If the reductions of A to B and of B to C take polynomial time, so does the reduction of A to C.

26.4.3 The NP-hard class

In 1971, Stephen Cook proved something extraordinary: *any* decision problem in NP can be reduced in polynomial time to SAT, the satisfiability problem. This means that, by definition, SAT is at least as hard as every NP problem. Problems that are as hard as every problem in NP are called **NP-hard**.

At the moment, no polynomial algorithm for SAT is known, but there's no proof that it can't exist. Imagine that someone finds a polynomial algorithm for SAT, thus proving that SAT is tractable. We saw earlier that if a problem A reduces in polynomial time to a tractable problem, then A must be tractable. So, since every NP problem reduces in polynomial time to SAT, if SAT turns out to be tractable, then every NP problem is tractable: $NP = P$. That's why I wrote in [Section 26.2.4](#) that SAT binds classes P and NP: to prove that $P = NP$ you only have to find a polynomial algorithm for SAT.

SAT was the first NP-hard problem to be found, but many others exist. Imagine that an NP-hard problem A reduces in polynomial time to problem B. Since any NP problem reduces in polynomial time to A, because A is NP-hard, then by transitivity of reduction it also reduces in polynomial time to B. Well, if there's a polynomial-time reduction from each NP problem to B, then B is NP-hard by definition. We have just proved the following statement.



Note: If an NP-hard problem A reduces in polynomial time to problem B, then B is NP-hard too.

This is the second example of using reduction to classify problems. Starting from SAT and using polynomial-time reductions, computer scientists have proven many problems to be NP-hard. I'll list a few in [Section 26.5.1](#). The earlier argument of why a polynomial algorithm for SAT proves that $NP = P$ can be applied to any other NP-hard problem.



Note: If someone finds a polynomial algorithm for *one* NP-hard problem, then $NP = P$.

26.4.4 The NP-complete class

An NP-hard problem is *at least* as hard as any NP problem, so it may not be in the NP class. When a problem is both NP-hard and in NP, the problem is **NP-complete**.

SAT is NP-complete because it is NP-hard (as proven by Stephen Cook) and in NP (as shown [before](#)). Another NP-complete problem is the [decision TSP](#). We've seen it's in NP, but the proof that it's NP-hard will be omitted.

The NP-complete problems are the hardest problems in NP, because they are at least as hard as every other NP problem.

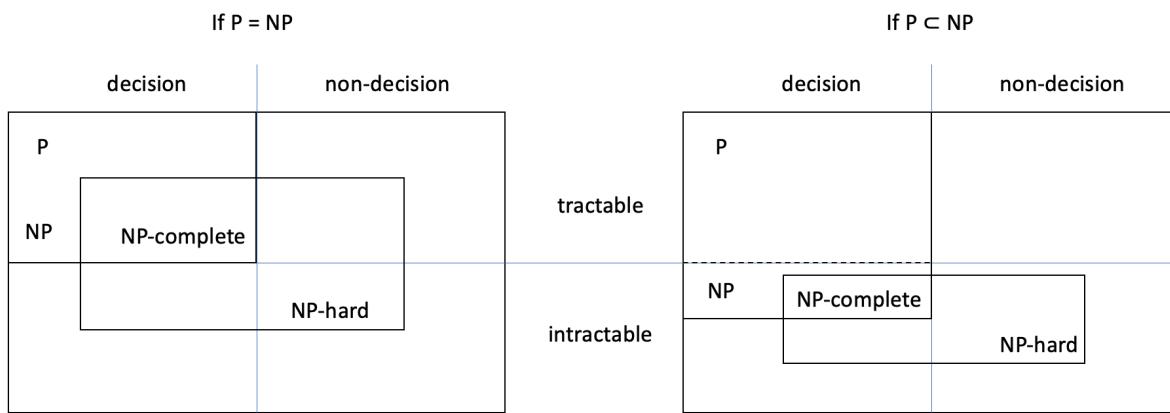
One striking property of NP-complete problems is that they all reduce in polynomial time to each other. The reason for this is as follows. Consider any two NP-complete problems A and B,

i.e. both are in NP and NP-hard. Since A is in NP and B is NP-hard, there's a polynomial-time reduction of A to B, by definition of NP-hardness. However, since B is in NP and A is NP-hard, there's also a polynomial-time reduction of B to A. So any two NP-complete problems reduce in polynomial time to each other. In a sense, all NP-complete problems are the same problem.

26.4.5 P versus NP

To conclude this section, let's see how the NP-hard and NP-complete classes fit with the other classes under both scenarios: $P = NP$ and $P \neq NP$. Here's a diagram showing the class relations.

Figure 26.4.2



In both cases, due to the definition of NP-completeness, the NP-complete class is the intersection of the NP and NP-hard classes.

The NP-hard class also includes non-decision problems because it has been proven that the travelling salesman problem and many other problems that don't have Boolean outputs are NP-hard.

We've seen earlier that if *any one* NP-hard problem is tractable, then *all* NP problems are tractable too. So when $P \neq NP$ (right-hand diagram), all NP-hard problems must be intractable, because otherwise we'd have $NP = P$. Since the NP-complete problems are a subset of the NP-hard problems, they must be intractable too when $P \neq NP$.

26.5 Theory and Practice

I close this chapter with some considerations about the theory and practice of algorithmic complexity.

26.5.1 Theory

Problem reduction is very important in helping to understand the nature of problems. If A reduces to B in polynomial time, then A can't be harder than B:

- if B can be solved in polynomial time, so can A, and
- vice versa, if A can't be solved in polynomial time, neither can B.

Polynomial-time reductions allow us to make general statements, like all NP-complete problems being equally hard and harder than any other NP problem.

Starting from SAT, computer scientists used polynomial-time reductions to classify hundreds of problems. I haven't shown any polynomial algorithms for the *travelling salesman*, *0/1 knapsack*, *maximal independent set* or *subset sum* problem for one simple reason: they all have been proven to be NP-hard.

One of the most fascinating aspects of complexity theory for me is that seemingly similar problems fall into different complexity classes. Here are some examples.

The **fractional knapsack** problem allows us to put a fraction of each item in the knapsack, e.g. when each item is in liquid or powder form. There's a simple greedy log-linear algorithm that solves this problem: go through the items from most to least profitable (value-to-weight ratio) and take as much as possible from each one until the knapsack is full. This means that the fractional knapsack problem is in P but the 0/1 knapsack problem is NP-hard. If we can put only 0% or 100% of each item in the knapsack, the problem becomes much harder.

Given a graph, the **all-pairs longest path** problem asks for a non-cyclic path with the most edges (if the graph is unweighted) or with the highest sum of weights (if it's weighted), among all pairs of nodes. This problem is NP-hard whereas the **all-pairs shortest path** problem is in P because we can use breadth-first search or Dijkstra's algorithm to solve the *single-source shortest paths problem* for each graph node, and then take the shortest path of them all.

Exercise 11.2.1 implicitly showed that the **primality** problem ('is a given positive integer prime?') can be reduced in polynomial time to the **factorisation** problem ('list the positive divisors of a given positive integer') because a number is prime if and only if it has exactly two factors. There's an *exponential algorithm* for the factorisation problem and therefore for the primality problem. This suggests that both problems are intractable, but we know that reducing A to B isn't necessarily the most efficient way of solving problem A. In fact, in 2002 three Indian scientists found a polynomial algorithm to check if a number is prime, proving that primality is in P. However, to this day it's not known whether factorisation is NP-hard, so finding a polynomial algorithm for it won't prove $P = NP$.

The **2-SAT** decision problem is a specialised version of SAT in which the input Boolean expression is a conjunction of disjunctions, where each disjunction has exactly two variables or their negations, e.g. '(A or B) and (not B or C)'. As you probably guessed, although SAT is NP-complete, 2-SAT is in P. The more specific problem can be solved much more efficiently than by applying exponential brute-force search, as for the general problem. Not all Boolean expressions can be written as a conjunction of two-variable disjunctions, so the general SAT problem remains important.

26.5.2 Practice

Reductions also have some practical value: if problem A reduces to problem B then *any* algorithm that solves B also solves A, via input and output transformations. Usually the obtained algorithm won't be the most efficient to solve A, but at least it's a start.

If somebody ever finds *one* polynomial algorithm for *one* NP-complete problem, then polynomial algorithms for the hundreds of known NP-complete problems can be immediately implemented, with the polynomial-time reductions used to prove that those problems are NP-complete.

Polynomial algorithms are often characterised as being efficient, because they are being contrasted with the non-polynomial exponential and factorial algorithms. In practice, an exponential algorithm may be efficient enough and a polynomial algorithm may be inefficient, because there are factors at play that are often ignored by the theory, to simplify the analysis.

Complexity analysis and the classification of problems is mostly based on worst-case scenarios, but in real life many inputs aren't worst cases, so theoretically inefficient algorithms may in practice solve most problem instances in a reasonable time. For example, backtracking can prune the search space effectively for many inputs even though there may be inputs for which all candidates have to be generated.

A further example is SAT solvers: they can routinely handle Boolean expressions with hundreds of variables and thousands of operators. Even though the worst case is exponential, it rarely occurs. In addition, SAT solvers return an interpretation that satisfies the expression, if there is one, i.e. they also return the certificate.

Complexity analysis predicts how the run-time grows for ever-increasing input sizes, but sometimes we're only interested in small problem instances. In such cases, an inefficient algorithm may be sufficient in practice. For example, supermarket delivery vans typically visit 15–20 customers before returning to the warehouse. A dynamic programming algorithm for the TSP can compute the best tour for graphs with that many nodes in a few seconds.

On the other hand, problems in P may actually have no practical algorithm. A polynomial algorithm with complexity $O(n^c)$ is inefficient if the exponent c is high or the constant factor (ignored by complexity analysis) is high. What exactly 'high' means depends again on the input size. If the input size n is in the thousands, then a cubic algorithm ($c = 3$) may be very slow.

Another example is the polynomial algorithm to test primality of a number n : it has complexity $O((\log n)^{12})$ and is therefore hardly used in practice. Primality testing is required for cryptography, where numbers can have more than 100 digits, i.e. $\log n > 100$, which makes this algorithm impractically slow.

When no exact algorithm is fast enough for large real-world problem instances, practitioners use heuristic algorithms that only provide approximate or probabilistic results. For example, primality is usually decided with an algorithm that gives the right decision with high (but not 100%) probability. And for large graphs, the TSP is solved with algorithms that return an approximate 'good enough' result. You have seen a greedy heuristic algorithm for the TSP in [Exercise 18.3.1](#).

So if you have an NP-complete or NP-hard problem at hand, first consider what input sizes you will need to cope with. If they are small, an exact exponential algorithm might do the job. Otherwise, look for a heuristic that gives a good approximation of the result or the right result with high probability.

If your problem can be reduced in polynomial time to SAT or the TSP, then implementing the input and output transformations yourself and using a good SAT or TSP solver might be your best bet.

26.6 Summary

Computer scientists have classified computational problems in many ways, based on the complexity of the algorithms that solve the problems.

26.6.1 Reductions

A **polynomial** algorithm has a worst-case complexity of the form $O(n^c)$, where n is the input size and c is a constant. For example, algorithms with complexity $O(n^3)$, i.e. that run in cubic time or less, are polynomial, whereas algorithms with exponential or factorial complexity aren't.

The **complexity of a problem** is the lowest worst-case complexity of all algorithms that solve the problem. To know a problem's complexity we must have an algorithm of that complexity and a proof that no other more efficient algorithm solves the problem.

Problem A **reduces** to problem B if there's an algorithm of the following form that solves A.

1. Transform the inputs of A into the inputs of B.
2. Use any algorithm for B.
3. Transform the obtained output for B into A's output.

Problem A is **at most as hard as** problem B, and B is **at least as hard as** A, if A reduces to B in polynomial time, i.e. if steps 1 and 3 take polynomial time.

A polynomial-time reduction of problem A to problem B can be used for three purposes:

- Obtain a (possibly inefficient) algorithm for unsolved problem A, given an algorithm for solved problem B.
- Prove that A is tractable, if we know that B is.
- Prove that B is NP-hard, if we know that A is.

26.6.2 Problem classes

The classes (sets) of problems covered in M269 are the following. (The first two are based on the output type: they aren't complexity classes.)

- Decision problems: those that have as output a Boolean value.
- Non-decision problems: all other problems.
- **Tractable** problems: those that can be solved by a polynomial algorithm.
- **Intractable** problems: those that can only be solved by non-polynomial algorithms.
- Class **P**: the tractable decision problems, i.e. those that can be solved in polynomial time.
- Class **NP**: the decision problems that can be verified in polynomial time.
- Class **NP-hard**: the problems that are at least as hard as every NP problem.
- Class **NP-complete**: the NP problems that are NP-hard.

A problem is in NP if there's a polynomial **verifier** algorithm that, for each input that leads to a true output (a 'yes' decision), takes the input and a **certificate**, and confirms that the decision is 'yes'.

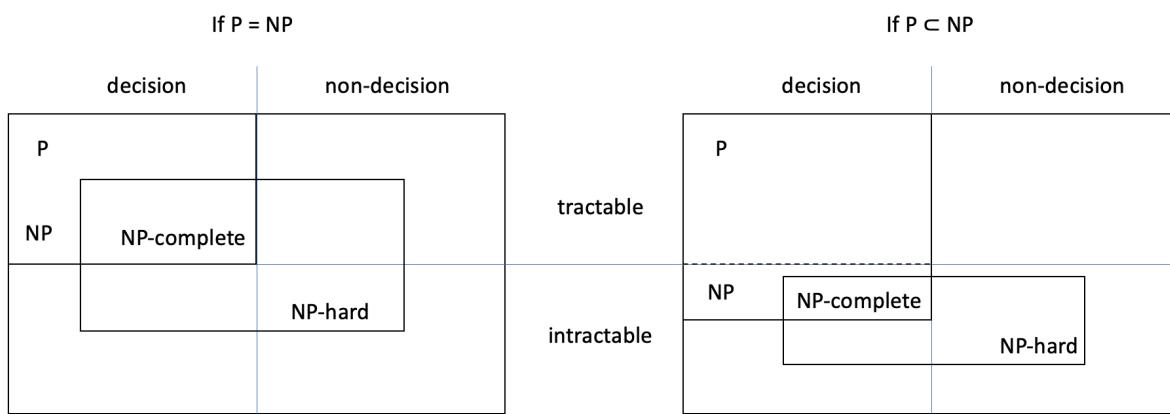
From the definitions it follows that:

- any NP problem reduces in polynomial time to any NP-hard problem
- all NP-complete problems reduce in polynomial time to each other
- all problems in P are also in NP: $P \subseteq NP$.

It's unknown if $P \subset NP$ or $P = NP$: this is the **P versus NP problem** or '**P = NP?**' question.

The relationships between the classes are as follows, for $P = NP$ and for $P \neq NP$:

Figure 26.6.1



In practice, a tractable problem may have no usable polynomial algorithm and an NP-complete or NP-hard problem may have a usable exponential algorithm; usability depends on the exponent of the complexity, the hidden constant factors and the real-world problem instances to be dealt with (their sizes and whether they are worst cases). For example, there's a polynomial algorithm to check if an integer is prime, but due to its large exponent ($c = 12$), it's not used in practice.

Heuristic algorithms are often used to handle large instances of NP-complete and NP-hard problems. They compute approximate answers or the correct answer with high probability.

26.6.3 Problems

The **satisfiability problem (SAT)** is:

Function: SAT

Inputs: *expression*, a string

Preconditions: the string represents a Boolean expression with only variables and logical operators

Output: *satisfiable*, a Boolean

Postconditions: *satisfiable* is true if and only if there are values for the Boolean variables that make *expression* true

The postcondition could be rephrased as ‘... if there’s an interpretation that satisfies *expression*’. An **interpretation** is an assignment of a Boolean value to each variable in the expression.

The **2-SAT** problem is like SAT, but the Boolean expression is a conjunction of disjunctions, with two variables or their negation per disjunction.

The **decision TSP** takes as input a complete weighted graph and a positive integer w . The Boolean output is true if and only if the graph has a tour with total weight w or less.

The **fractional knapsack** problem is like the 0/1 knapsack problem, but the solution can involve taking only part of each item. The output is therefore not just the subset of items put in the knapsack but also the percentage of each item.

The **all-pairs shortest (or longest) path** problem asks for a shortest (or longest non-cyclic) path in a graph, between any two nodes. If the graph is unweighted, the path has the fewest (or most) edges; if the graph is weighted, the path has the lowest (or highest) sum of weights.

Seemingly similar problems can be in different complexity classes:

Class	Problems
NP-hard	(decision) <i>TSP</i> , <i>0/1 knapsack</i> , <i>subset sum</i> , <i>maximal independent set</i> , all-pairs longest path, SAT
NP-complete	SAT, decision TSP
tractable	all-pairs shortest path, fractional knapsack, 2-SAT, <i>primality</i>
P	2-SAT, primality

Most problems presented in M269 are tractable.

CHAPTER 27

COMPUTABILITY

We started M269 with problems that can be solved with simple and efficient algorithms, like linear search. Problems became increasingly complicated, but most could still be solved efficiently with clever algorithms and data structures, like binary search and heaps. Even for the most expressive data type – graphs – most algorithms had quadratic complexity or better.

Nevertheless, for problems like the travelling salesman and the 0/1 knapsack, even techniques like backtracking or dynamic programming didn't lead to better than exponential algorithms. The previous chapter showed that it's unlikely polynomial algorithms will be found for these problems.

This chapter concludes M269 with the hardest of all problems: those that have no algorithmic solution at all: not now, not ever.

This chapter supports these learning outcomes:

- Know about the limits of computation and their practical implications – this chapter introduces three non-computable problems that have practical implications.
- Understand the Turing Machine model of computation – this chapter introduces the computational model that is the basis of all formal definitions.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#), and check the TMAs for what is assessed.

27.1 Turing machine

In 1936, mathematician Alan Turing captured the intuitive notion of algorithms in a formal way. There were no programmable analogue or electronic computers in those days, so he proposed a conceptual machine that could carry out computations.

The following is one of many possible definitions of **Turing machines**. As we shall see later, it doesn't really matter which definition we adopt.

27.1.1 Definition

The ‘hardware’ of the Turing machine consists of a **tape** divided into **cells**. Each cell holds one **symbol**. The symbols are anything that’s convenient for the problem at hand: bits (0 and 1), characters, 64-bit integers, etc. The special **blank** symbol indicates that a cell is empty.

The tape is infinite: it has a start but no end. The machine has a **head** that is over one cell at a time. The head may move left or right, one cell at a time. The head can read the symbol in the cell it’s over (the **current symbol**) and write a symbol into that cell. Initially, the head is at the start of the tape. If at any time the head moves to the left of the first cell, the machine stops.

Since the tape is infinite, only part of it contains the data; the rest is an infinite sequence of blanks. The **input** is the initial data on the tape: the zero or more symbols before the infinite blank sequence. When the machine stops executing, the **output** is the symbols from the head onwards until the infinite blank sequence starts.

When we write an algorithm in English or implement it in a programming language, the problem definition tells us how many inputs and outputs there are, their types and all the conditions they must satisfy. Likewise, problem statements for Turing machines must tell us the initial and final content of the tape, what symbols are used and where the head is when the machine stops. You will see an example in a minute.

As for the ‘software’ of a Turing machine, it defines when and how to move the head and which symbol to write. This depends on the **current state** of the machine. There’s a finite set of possible states and we must indicate which one is the **initial state**.

The program for the machine is of the form

1. put head on the start of the tape
2. let current state be the initial state
3. repeat forever:
 1. if current state = ... and current symbol = ...:
 1. write symbol ...
 2. move the head left or right or not at all
 3. let current state be ...
 2. otherwise if current state = ... and current symbol = ...:
 1. write symbol ...
 2. move the head left or right or not at all
 3. let current state be ...
 3. ...
 4. otherwise:
 1. stop

There's at most one if-statement per state–symbol pair and so the order of the if-statements doesn't matter. For n symbols and m states, there are at most $n \times m$ if-statements. If there's no if-statement for the current symbol and state (step 3.4 above), the machine stops.

Turing machines use the three algorithmic ingredients: iteration (repeat forever), selection (the if-statements) and sequence of instructions.

The if-statements can be represented more compactly as a **transition table** that has one row per state and one column per symbol, including the blank symbol. If there's an if-statement for a particular state and symbol, then the table entry for the corresponding row and column indicates the **execution step**: the symbol written (which can be the same as the symbol read), the movement of the head (left, right or stay) and the new state (which can be the same as the current state).

A **configuration** is made of the current state, the position of the head and the content of the tape. Each execution step makes the Turing machine transition from one configuration to the next.

Let's see an example Turing machine.

27.1.2 Parity bit

One method to detect if binary data was corrupted during transmission is for the sender to append a bit (0 or 1) to the message, so that the number of ones is even. The added bit is called the parity bit. The receiver checks the number of ones: if it's odd, it asks for the message to be sent again, otherwise it discards the parity bit.

For example, if the message is 011 then the parity bit is 0 and the sender transmits 0110. If the receiver gets 0111 then it knows something is wrong (here, the parity bit changed) and asks for a re-transmission of the message. This method can't detect all errors. For example, if the message sent is 011011 and the message received is 100111, the errors aren't detected because the number of 1s remains even.

Here's the problem description for a Turing machine.

The tape initially has a blank followed by a possibly empty sequence of 0s and 1s.
Add a 0 or 1 at the end, so that the number of 1s is even. Return the head to the start of the tape.

As mentioned before, the description must include the needed symbols ('0s and 1s'), the input ('a blank followed by ...'), what to do ('add a 0 or 1 at the end') and where the head ends up ('return the head to the start') in order to know what the output (the data to the right of the head) is.

When providing input and output examples, we only represent the data as a finite sequence of symbols, omitting the infinite sequence of blank cells. The above example can be restated as: if the input is (blank, 0, 1, 1), the output is (blank, 0, 1, 1, 0).

To solve the problem, we should first outline the algorithm, before writing the transition table. Here's a first rough outline.

Move the head to the right, over the 0s and 1s, keeping track of whether the number of 1s is even. When a blank is reached, write a 0 or 1 accordingly. Move the head to the left, over the 0s and 1s. Stop when a blank is reached.

This outline is incomplete because it doesn't explain how to determine if there's an odd or even number of 1s. If a Turing machine had variables, we could count the 1s, but since it hasn't, we must instead think in terms of states.

All we need to keep track of, as we process the input, is the parity of the 1s, so let's create two states called 'odd' and 'even'. We don't change state when reading a zero, but when we read a 1, we change the state from 'odd' to 'even' or vice versa. When we read a blank, we write a 0 if in the 'even' state, otherwise a 1. We then get into a new state, let's call it 'back', where we simply move to the left until reading a blank.

Finally, we must think of the initial state. Initially, we haven't read any bits, so we should start in the 'even' state. However, the first symbol is a blank. We already have an action for a blank in the 'even' state: write 0 and start moving back. The only way for a machine to distinguish the start blank from the end blank is by being in two different states. We need therefore an extra initial state, let's call it 'start'.

Here's a better outline of the algorithm, detailing the actions and providing some of the rationale.

1. Begin in state 'start'.
2. Skip the initial blank and change to state 'even' because zero 1s have been read so far.
3. Move right over any 0s and 1s, switching between the 'even' and 'odd' state when reading a 1.
4. When reading a blank, write 0 if the state is 'even', otherwise write 1, because there's an odd number of 1s and so we need an extra 1.
5. After writing the parity bit, move left and change to the 'back' state.
6. Move left, skipping any 0s and 1s.
7. Don't do anything when reading a blank: this stops the machine with the head at the start of the tape.

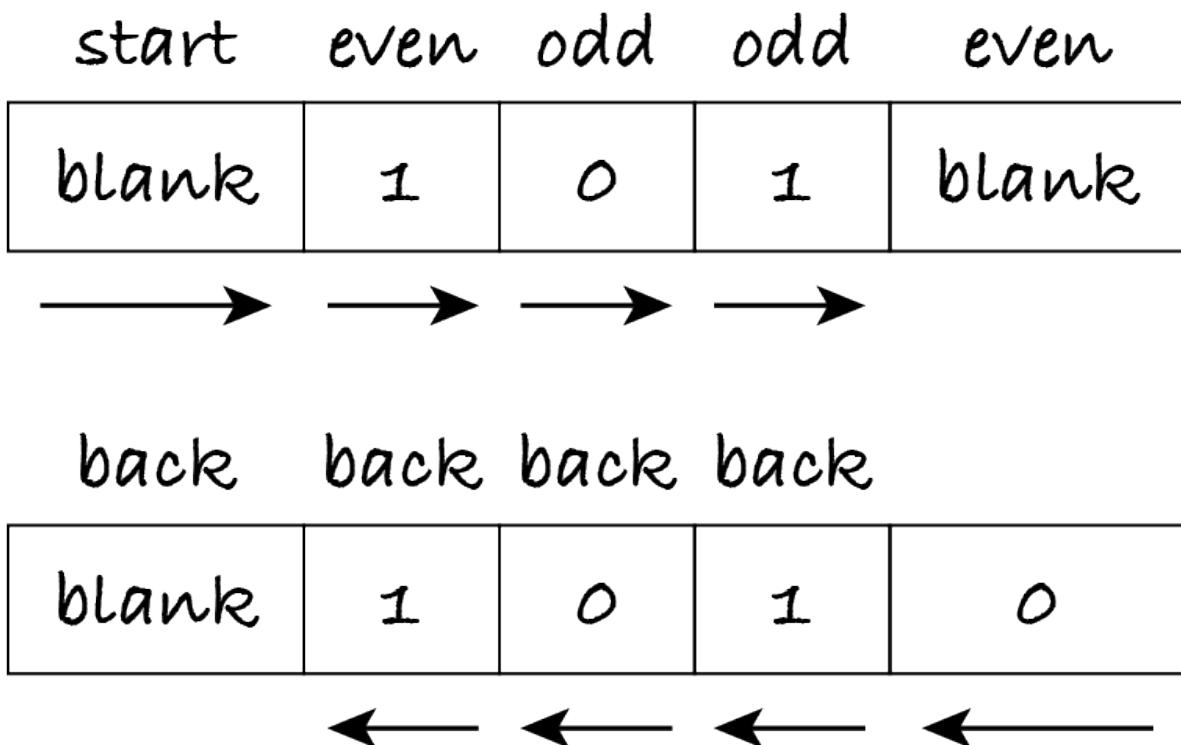
Notice that an outline doesn't have to spell out all details. For example, I use the word 'skip' to indicate that the head moves over the symbols without changing them. However, when writing the algorithm as a transition table, we must always indicate the symbol written, the head movement and the new state.

State	blank	0	1
start	write blank, right, even		
even	write 0, left, back	write 0, right, even	write 1, right, odd
odd	write 1, left, back	write 0, right, odd	write 1, right, even
back		write 0, left, back	write 1, left, back

As the table shows, except when reaching a blank in the 'even' or 'odd' state, we always write the symbol that has been read, to keep the tape unchanged.

The next figure shows how the machine processes input (blank, 1, 0, 1). The figure shows above each symbol the current state and below the symbol how the head moves. The new state, after moving the head, is the state above the next cell.

Figure 27.1.1



The machine begins in state ‘start’, skips the blank by moving the head right and enters state ‘even’. It continues reading symbols and moving right, changing state after reading a 1.

When it reaches the blank after the bit sequence, it writes a 0 because it’s in state ‘even’, moves the head left and enters state ‘back’. It keeps moving the head left in that state until the first blank is reached. There’s no entry in the transition table for state ‘back’ and the blank symbol, so the machine stops.

27.1.3 Implementation

This subsection explains how I implemented Turing machines and the functions I created for you to write, run and test Turing machines. All functions are in file `m269_tm.py`, so let’s run it to load those functions.

```
[1]: %run -i ../m269_tm
```

The Turing machine simulator will take a transition table, an input tape and an initial state, and execute steps until there’s no transition for the current state and symbol. To simplify things, we adopt the following convention.



Note: In M269, the initial state of a Turing machine is always called ‘start’.

The simulator will be implemented as a Python function with two inputs (a transition table and a tape) and one output (the tape from the head’s final position onwards).

There are many ways to represent transition tables in Python. The simplest, in my view, is a map (Python dictionary) of state–symbol pairs to symbol–movement–state triples: given the current state and symbol, the map tells us what symbol is written, how the head moves and what's the new state.

I will represent the tape as a list, with one symbol per cell. The blank symbol is represented with `None`. The current position of the head is a natural number, indexing the list. The movement can thus be represented as how the current index changes: -1 to move the head left, $+1$ to move it right and 0 if it doesn't move.

Here's the transition table for the parity problem, written as a Python dictionary. For readability, it uses constants `LEFT` (-1), `STAY` (0), `RIGHT` ($+1$) defined in `m269_tm.py`. A dictionary's key–value pairs can be given in any order. One possibility is to follow the table I wrote earlier, row by row, to make sure no state–symbol combination is forgotten.

```
[2]: parity = {
    # (state now, symbol read): (symbol written, move, next state)
    ('start', None):      (None, RIGHT, 'even'),
    ('even', None):        (0, LEFT, 'back'),
    ('even', 0):           (0, RIGHT, 'even'),
    ('even', 1):           (1, RIGHT, 'odd'),
    ('odd', None):         (1, LEFT, 'back'),
    ('odd', 0):            (0, RIGHT, 'odd'),
    ('odd', 1):            (1, RIGHT, 'even'),
    # ('back', None):      stop
    ('back', 0):           (0, LEFT, 'back'),
    ('back', 1):           (1, LEFT, 'back')
}
```

An alternative order follows the algorithm.

```
parity = {
    ('start', None):      (None, RIGHT, 'even'),
    # skip 0s and 1s, switching state on each 1
    ('even', 0):           (0, RIGHT, 'even'),
    ('even', 1):           (1, RIGHT, 'odd'),
    ('odd', 0):            (0, RIGHT, 'odd'),
    ('odd', 1):            (1, RIGHT, 'even'),
    # when reading a blank, write the parity bit and start moving back
    ('even', None):        (0, LEFT, 'back'),
    ('odd', None):         (1, LEFT, 'back'),
    # skip all 0s and 1s, stop when reaching the first blank
}
```

(continues on next page)

(continued from previous page)

```

('back', 0):      (0, LEFT, 'back'),
('back', 1):      (1, LEFT, 'back')
}

```



Note: In a TMA, you can list the transitions in either of these orders, as you prefer.

Before running a machine, I recommend you check it with `check_tm`, a function that can spot small mistakes, like mistyping a state name. The function needs the sets of input and output symbols, to check among other things that each input symbol is read by at least one transition (otherwise the machine wouldn't handle all possible input tapes). In this case, the input and output symbols are the same: 0 and 1.

```
[3]: PARITY_IN = {0, 1}
PARITY_OUT = PARITY_IN

check_tm(parity, PARITY_IN, PARITY_OUT)
OK: the transition table passed the automatic checks
```

All's well with this machine. Here's a bogus machine to illustrate some errors that `check_tm` spots.

```
[4]: wrong_parity = {
    # wrong name for initial state, which becomes unreachable
    ('begin', None):      (None, RIGHT, 'even'),
    # state, symbol and movement in wrong order, in both tuples
    (None, 'even'):        (0, 'back', LEFT),
    # missing movement
    ('odd', 1):            (1, 'even'),
    # typo in state name, which leads to another unreachable state
    ('eben', 1):           (0, RIGHT, 'even'),
    # symbol 2 is read (expected on the tape) but is not in input
    # nor written by another transition
    ('even', 2):           (2, RIGHT, 'even'),
    # symbol 3 is written but not used in output or by another
    # transition
    ('even', 1):           (3, RIGHT, 'odd')
}

check_tm(wrong_parity, PARITY_IN, PARITY_OUT)
ERROR in (None, 'even'): None is not a state name (string)
ERROR in (0, 'back', -1): back is not STAY, LEFT or RIGHT
ERROR in (1, 'even'): not a tuple (symbol, movement, state)
```

(continues on next page)

(continued from previous page)

```
ERROR: no state named 'start'  
ERROR: no transitions to states {'begin', 'eben'}  
ERROR: no transitions read input symbols {0}  
ERROR: symbols {2} are read but no other transitions write them  
WARNING: symbols {2, 3} are written but no other transitions read  
→them
```

Did you spot the error I didn't mention in the comments? (No transition reads 0.) Such small errors can be difficult to find and fix just by testing the machine, hence it's always best to check the transition table first. The checking function can't guarantee the machine is correct, but at least it can catch some mistakes.

Now we can run the machine with function `run_tm`. It looks up the dictionary with the current state and symbol and carries out the associated actions: write a symbol, move the head, change the state. It keeps doing this until it finds no entry for the current state–symbol pair.

The Turing machines you will be asked to write in M269 are simple and will have short inputs. If your machine hasn't stopped after 100 steps, it probably entered an infinite loop, and the simulator will stop.

Since the tape is only infinite to the right, the simulator also stops if the head goes left past the first position.

Once the simulator stops, it returns the content of the tape from the head's position onwards. That's the output of the machine. Here's the output for the example input given in the figure above.

```
[5]: run_tm(parity, [None, 1, 0, 1])  
[5]: [None, 1, 0, 1, 0]
```

To see what's happening step by step, the simulator can print out the configurations it is going through. Each configuration is printed in the form

```
counter current state [symbols left of the head] current symbol  
→ [symbols right of the head]
```

where the counter starts at zero for the initial configuration. The configurations are printed by enabling debugging mode through a third, Boolean argument.

```
[6]: run_tm(parity, [None, 1, 0, 1], debug=True)  
0 start [] None [1, 0, 1]  
1 even [None] 1 [0, 1]  
2 odd [None, 1] 0 [1]  
3 odd [None, 1, 0] 1 []  
4 even [None, 1, 0, 1] None []  
5 back [None, 1, 0] 1 [0]  
6 back [None, 1] 0 [1, 0]
```

(continues on next page)

(continued from previous page)

```

7 back [None] 1 [0, 1, 0]
8 back [] None [1, 0, 1, 0]
[6]: [None, 1, 0, 1, 0]

```

We can use test tables to run a machine with several inputs and compare the actual outputs with the expected ones. The test table just includes the input tape, the debug parameter and the expected output for each test. Debugging is initially off but we can turn it later on for tests that fail.

```

[7]: parity_tests = [
    # name,      input,      debug,      output
    ('no bits', [None], False, [None, 0]),
    ('just 0',   [None, 0], False, [None, 0, 0]),
    ('just 1',   [None, 1], False, [None, 1, 1]),
]

```

I recommend checking the test table with `check_tm_tests` before running the tests. This function also needs the input and output symbols, to check that each test only uses those symbols.

```

[8]: check_tm_tests(parity_tests, PARITY_IN, PARITY_OUT)
OK: the test table passed the automatic checks.

```

Finally, we can run the tests with the new function `test_tm`, which has two parameters: the machine (transition table) to test, and the test table.

```

[9]: test_tm(parity, parity_tests)
Tests finished: 3 passed (100%), 0 failed.

```

The function takes an optional third parameter, a Boolean that controls whether the tests' names are printed as they're executed. This helps tracking what's happening, especially when debugging several tests.

```

[10]: parity_tests_debug = [
    # name,      input,      debug,      output
    ('no bits', [None], True, [None, 0]),
    ('just 0',   [None, 0], False, [None, 0, 0]),
    ('just 1',   [None, 1], True, [None, 1, 1]),
]

test_tm(parity, parity_tests_debug, show_tests=True)

Running test no bits...
0 start [] None []
1 even [None] None []
2 back [] None [0]
Running test just 0...

```

(continues on next page)

(continued from previous page)

```
Running test just 1...
0 start [] None [1]
1 even [None] 1 []
2 odd [None, 1] None []
3 back [None] 1 [1]
4 back [] None [1, 1]
Tests finished: 3 passed (100%), 0 failed.
```

Now over to you, for some practice.

27.1.4 Checking passwords

We want a machine that decides whether the tape contains a valid password: a password with at least one letter and at least one digit. To represent all English letters and all 10 digits, we will use only two symbols: characters ‘a’ and ‘0’.

The input is a possibly empty sequence of those two characters. The machine should write at the end of the sequence the corresponding Boolean, keep the head over it and stop. The set of symbols thus includes two further symbols, the Boolean values, to represent the output. For example, if the input is (‘a’, ‘a’), then the output is (false). Remember that in this case the output is the sequence of symbols from the head to the right. The final tape is actually (‘a’, ‘a’, false).

Exercise 27.1.1

Outline an algorithm for the Turing machine. Don’t forget to indicate the states.

Hint Answer

Exercise 27.1.2

Complete the transition table below. I wrote the first two transitions for you.

```
[11]: VALID_IN = {"0", "a"}
VALID_OUT = {True, False}

is_valid = {
    ('start', 'a'):      ('a', RIGHT, 'letter'),
    ('start', '0'):      ('0', RIGHT, 'digit'),
    # add your transitions here
}

check_tm(is_valid, VALID_IN, VALID_OUT)

is_valid_tests = [
    # case,           input,            debug,   output
    ('empty pwd',     [],             False,   [False]),
    ('no digits',     ['a', 'a'],     False,   [False]),
    ('no letters',    ['0'],          False,   [False]),
```

(continues on next page)

(continued from previous page)

```
('both',          ['0', 'a', '0'],  False,   [True])  
]  
  
check_tm_tests(is_valid_tests, VALID_IN, VALID_OUT)  
  
test_tm(is_valid, is_valid_tests)
```

Remember that you can see the step-by-step configurations by setting `debug` to `True`. In that case you may wish to add `True` as a final argument to `test_tm`.

Hint Answer

Exercise 27.1.3

Imagine the input may include all 26 English letters, both in uppercase and lowercase, and all 10 digits. How would you change the transition table to check that the input is a valid password, i.e. has at least one letter and at least one digit? Don't write the table: just describe what additional entries would be needed. (The table would be rather long. Can you estimate how many additional entries are needed?)

Hint Answer

Exercise 27.1.4

Now imagine we require a valid password to also include at least one of the three punctuation marks ‘.’, ‘!?’ or ‘?’ Which states would you need?

Hint Answer

27.2 The Church–Turing thesis

In this section, I discuss the importance of Turing machines.

27.2.1 Computational models

While a programming language like Python eases the implementation of algorithms, it's not a good medium to define what algorithm and complexity mean. A programming language simply has too many constructs, making definitions more complicated. For example, we had to define ‘input size’ for each data type: for integers, it's the number of digits; for sequences and sets, it's the number of elements; for graphs, it's the number of nodes and edges. We also had to make assumptions about the complexity of each operation, like numeric operations on 64-bit numbers taking constant time, in order to analyse the complexity of an algorithm.

In contrast, a Turing machine is so simple that it became the main computational model, on which all concepts are based. A Turing machine defines precisely the concepts of input, output, algorithm (the transition table) and executing an algorithm (what a computational step is and when to stop). This in turn allows derived concepts to also be precisely defined:

- An algorithm is correct, i.e. solves a given problem, if it stops, for every input that satisfies the preconditions, with an output that satisfies the postconditions.
- A problem is **computable** if there's an algorithm that solves it.
- The size of the input is the number of symbols until the start of the infinite sequence of blanks.
- The run-time of an algorithm on a given input is the number of steps executed until it stops.
- The complexity of an algorithm is the growth rate of the number of steps in terms of the number of input symbols.

In summary, the formal definition of algorithm is what can be written as a transition table for a Turing machine, and all other definitions (computability, complexity, classes P and NP, etc.) are based on it.



Note: Turing machines are a formal model of computation: they enable a precise definition of algorithm, complexity and computability.

You may be wondering if basing the notion of computation on such simple conceptual devices is too restrictive. Is the Turing machine a good model of what modern computers do? Can they compute things the Turing machine can't?

27.2.2 Universal models

In the 1936 paper where Turing introduces his conceptual machine, he provides an argument (not a proof!) for why such a machine can mimic what computers can do. In those days, before electronic computers existed, the word 'computer' still meant 'a human who does mathematical calculations'.

Turing's argument was in essence as follows. When solving a mathematical problem, humans read and write symbols on sheets of paper and they have a mental state of what to do next. For example, when adding two numbers, they look at each digit, mentally carrying over if necessary, and write the result one digit at a time. Humans can only process a finite number of symbols and can only remember a finite number of things. They can use more sheets of paper as necessary. While the two-dimensional nature of paper sheets is convenient to put symbols above or under each other, it's not really essential to the computation itself. For example, the addition of two numbers can also be carried out horizontally.

Therefore, a machine that has an infinite tape, a head that moves back and forth to read and write symbols, and a finite number of states, should be able to compute anything a human (and, nowadays, electronic computers) can. This is known as the **Church–Turing thesis**: anything that can be computed, can be computed by a Turing machine. The statement is a hypothesis, not a theorem, because it can't be proven: it states that whatever people would informally agree to call an algorithm can be written as a transition table for the Turing machine.

Earlier, mathematician Alonzo Church defined the **lambda calculus**, a computational model

based on functions. It has been proven that the lambda calculus and Turing machines are equivalent, in the sense that what can be computed with one can be computed with the other. Several variants of Turing machines were also proven to be equivalent. For example, a Turing machine with multiple tapes, each with one head that can move independently of the other heads, can be simulated with a single-tape machine. No computational model has yet been invented that can solve more problems than Turing machines, and this gives further credibility to the Church–Turing thesis.

A modern computer, with multiple CPU cores, gigabytes of RAM and a large disk, cannot solve any problem the humble Turing machine can't. Very roughly, the reason is as follows.

Any program (in Python, Java, etc.) is translated to machine code, so that the CPU can execute it. Machine code provides a limited number of simple instructions: increment the value in a register, add the values in two registers, fetch a value from a RAM address and put it in a register, etc. If we store the binary content of each memory (register, cache, RAM, disk) in a separate tape with symbols 0 and 1, then each machine code instruction can be implemented with a transition table that moves the heads of the affected tapes. The transition tables are quite large: you've seen how many entries were needed for a simple problem like checking a string is a valid password. However, the point is that a Turing machine that simulates the execution of CPU instructions *can* be written.



Info: How computers process instructions is explained in TM112 Block 1 Part 3.

Since multi-tape machines are equivalent in computational power to single-tape machines, it follows that any algorithm written in Python (or any other language) and executed by a modern computer can be written as a transition table and executed by a Turing machine.

If any computable problem can be solved with a Turing machine, it means that it can be solved with a Python function that takes a single input of type `list` and has only two variables: a string with the current state and an integer with the index of the list element currently processed. The reason why two variables suffice to write any algorithm is that we can encode additional variables as states or in the list.

When solving the *parity bit* problem, we used additional states. Instead of having two Boolean variables that remember whether a letter or a digit occurred, we represented their four possible values with four states: 'start' (no letter or digit seen), 'letter', 'digit' and 'both'.

Now let's see an example of storing the additional variables in the tape rather than as additional states.

27.2.3 Length of string

The problem to be solved is to compute the length of a string:

Given a tape with zero or more letters 'a', add 0s and 1s at the end so that they represent, in reverse binary form, the number of letters. When the machine stops, the head is over the left-most digit. For example, if the input is ('a', 'a'), the output

is (0, 1) because the number of letters (namely, two) is 10 in binary. The input should be preserved, i.e. it should not be overwritten by other symbols.

Having the binary digits in reverse order allows us to grow the number to the right as we count letters, without overwriting the string.

To solve this problem, when reading an ‘a’, we must move the head right, increment the binary number and return the head to where it was, so that we can read the next ‘a’ without inadvertently skipping any letters.

We thus need to mark the position of the last ‘a’ counted, so that the head can return to that position. The best way to do that is to introduce a new symbol that marks the position. I will use ‘X’ as the symbol (because X marks the spot), but any symbol different from ‘a’, blank, 0 and 1 will do.

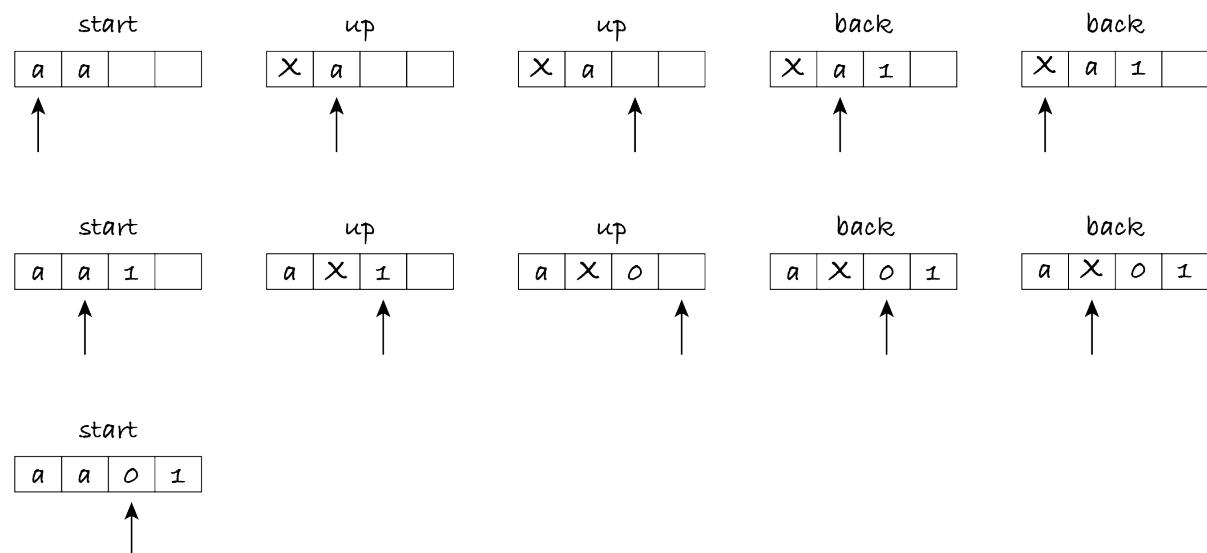
We need two states besides ‘start’. One state is to increment the counter, let’s call it ‘up’; the other is to return to the last ‘a’ read, let’s call it ‘back’.

In the ‘up’ state, the head moves right skipping all letters. If it finds a 0 or blank, it writes 1 (one more letter was read) and starts moving back. If it finds a 1, it writes 0 and moves right, staying in the ‘up’ state to carry over the bit and increment the rest of the binary number.

In the ‘back’ state, the head moves left, skipping all symbols until it reads ‘X’. At that point, the ‘X’ is replaced by ‘a’ (to not change the input), the head moves right and we return to state ‘start’, because we’re now going to repeat the same process to count the next letter.

The next figure shows the configurations, i.e. state, tape and head position, the machine goes through to process input (‘a’, ‘a’). The first and second rows of the diagram show, from left to right, the five configurations to count each ‘a’. The third row shows the final configuration: the machine is again in the ‘start’ state and the head is over a digit. This means all letters were processed and the machine stops.

Figure 27.2.1



Here’s the transition table.

```
[1]: %run -i ./m269_tm

LENGTH_IN = {"a"}
LENGTH_OUT = {0, 1}

length = {
    # (state, symbol): (symbol, head, state)
    # if 'a', mark the position and start incrementing counter
    ('start', 'a'): ('X', RIGHT, 'up'),
    # if empty string, write 0 and stop
    ('start', None): (0, STAY, 'start'),

    # before incrementing, skip all letters
    ('up', 'a'): ('a', RIGHT, 'up'),
    # if bit is zero: increment and return to marked position
    ('up', 0): (1, LEFT, 'back'),
    # if bit is one, carry over: keep incrementing
    ('up', 1): (0, RIGHT, 'up'),
    # end of binary number: increment and go back
    ('up', None): (1, LEFT, 'back'),

    # to return to marked position, skip all digits and letters
    ('back', 0): (0, LEFT, 'back'),
    ('back', 1): (1, LEFT, 'back'),
    ('back', 'a'): ('a', LEFT, 'back'),
    # restore marked position to 'a'; start again with next letter
    ('back', 'X'): ('a', RIGHT, 'start')
}

check_tm(length, LENGTH_IN, LENGTH_OUT)

length_tests = [
    # case,      input,          debug, output
    ('empty',    [],            False, [0]),
    ('one',      ['a'],         False, [1]),
    ('two',      ['a', 'a'],    False, [0, 1]),
    ('three',    ['a', 'a', 'a'], False, [1, 1]),
]
check_tm_tests(length_tests, LENGTH_IN, LENGTH_OUT)

test_tm(length, length_tests)

OK: the transition table passed the automatic checks
OK: the test table passed the automatic checks.
Tests finished: 4 passed (100%), 0 failed.
```

Don't forget to replace `False` with `True` if you want to see the steps for any of the tests.

In summary, although Turing machines are very constrained, we can write any Python function as a (possibly very complicated) transition table, by representing variables as part of the states or part of the tape's content.

Exercise 27.2.1

If the input symbols were all the 26 lowercase and 26 uppercase letters, instead of just ‘a’, what changes would you make to the transition table? For this exercise you don’t have to preserve the input: you can overwrite it with marker symbols.

Hint Answer

Exercise 27.2.2

Do again the previous exercise, but this time preserving the input.

Hint Answer

27.3 Static analysis

The next section will show three problems that can't be solved by any algorithm. The reason is because those problems are about functions, not data (numbers, strings, graphs, etc.). So let's first look more closely at that kind of problem in this section. Since algorithms for Turing machines are unwieldy to write, let's return to Python.

27.3.1 Functions on functions

Consider the following problem. It takes as input a function, not a value of some data type.

Function: get info

Inputs: f , a Python function

Preconditions: true

Output: $text$, a string

Postconditions: $text$ is the header and docstring of f

This problem is solved by the built-in function `help`, introduced in [Section 2.6.1](#).

```
[1]: help(len)

Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Another example of functions being inputs was given in [Section 14.1.4](#): Python's `sort` method and `sorted` function can take as argument a function that for each item computes the key to be used for sorting. The example was to sort cards by suit or by value.

```
[2]: %run -i ./m269_sorting      # defines key functions for sorting
sorted(["2S", "AS", "2D", "AD"], key=suit)  # diamonds, then spades
[2]: ['2D', 'AD', '2S', 'AS']
```

```
[3]: sorted(["2S", "AS", "2D", "AD"], key=value)  # aces, then twos
[3]: ['AS', 'AD', '2S', '2D']
```

Function `help` can also be applied to our functions, not just those built-in.

```
[4]: help(suit)  # replace suit with value to see what value does
Help on function suit in module __main__:
suit(card: str) -> str
    Return the second character of the card.

    Preconditions: card has two characters;
    the first is 'A', '2' to '9', 'T', 'J', 'Q' or 'K'
    the second is 'C', 'D', 'H' or 'S'
```

A further example of a ‘function on functions’ is `test`, used throughout this book: it takes as inputs a test table and the function to be tested.

As a final example, consider retrieving the function’s source code, including the header and docstring.

Function: `get_code`

Inputs: f , a Python function

Preconditions: the source code of f is available

Output: $text$, a string

Postconditions: $text$ is the source code of f

This problem is solved by function `getsource` in module `inspect`. It doesn’t work for built-in functions like `len` because their source code isn’t available, but it works for code we wrote.

```
[5]: from inspect import getsource
print(getsource(suit))  # replace suit with value if you wish
def suit(card: str) -> str:
    """Return the second character of the card.

    Preconditions: card has two characters;
    the first is 'A', '2' to '9', 'T', 'J', 'Q' or 'K'
    the second is 'C', 'D', 'H' or 'S'"""

    (continues on next page)
```

(continued from previous page)

```
"""
return card[1]
```

You can also look at the source code of imported functions, like `test`, but it's likely they will contain Python constructs you haven't learned.

Python provides useful functions that operate on functions, like `help` and `getsource`, but even more useful is the ability to write such functions ourselves.

27.3.2 Writing functions on functions

When we write an assignment like `x = 5`, the Python interpreter allocates some memory and stores there an object representing 5. The object not only contains the value 5 but also information that it's of type `int`. Finally, the interpreter creates pointer `x` to associate name `x` to the memory address where the created object is stored.

The interpreter must store the object's type so that it knows what operations are valid. When we write `x + 1`, the interpreter follows the `x` pointer to find the associated object, retrieves the type, which is `int`, and confirms the addition operation is defined. Whereas if we write `len(x)`, the interpreter detects that because `x` is an integer, it has no operation `len` and reports a type error.

Similarly, when we write `def suit(card: str) ...`, the interpreter allocates memory, stores the function there together with information that this object is of type `Callable` ([Section 14.3.2](#)) and creates pointer `suit`, pointing at the stored function. When we write `suit('2S')`, the interpreter follows the `suit` pointer, confirms the associated object is of type `Callable`, i.e. a function, and then passes the string to the function and executes it.

Therefore, there's nothing magic about `help`, `getsource` and `sorted`. They simply take as input a pointer to a `Callable` object and then extract the docstring or source code of the stored function or, in the case of `sorted`, execute the function on each item to obtain its sorting keys. Our sorting functions in Chapter 14 do exactly that: they take a key function as input and apply it to the items to sort them. Here again is our [`insertion sort`](#) code, without docstring and comments.

```
[6]: from typing import Callable

def insertion_sort(items: list, key: Callable) -> None: # noqa: D103
    for first_unsorted in range(1, len(items)):
        to_sort = items[first_unsorted]
        the_key = key(to_sort)
        index = first_unsorted
        while index > 0 and key(items[index - 1]) > the_key:
            items[index] = items[index - 1]
            index = index - 1
        items[index] = to_sort
```

Notice the calls `key(to_sort)` and `key(items[index - 1])` to the function `key` given as an argument.

27.3.3 Navel gazing

If a function takes *any* function as input, then it can be applied to itself.

```
[7]: help(help)

Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
| Define the builtin 'help'.
|
| This is a wrapper around pydoc.help that provides a helpful_
| message
| when 'help' is typed at the Python interactive prompt.
|
| Calling help() at the Python prompt starts an interactive help_
| session.
| Calling help(thing) prints help for the python object 'thing'.
|
| Methods defined here:
|
| __call__(self, *args, **kwds)
|     Call self as a function.
|
| __repr__(self)
|     Return repr(self).
|
| -----
|
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables
|
| __weakref__
|     list of weak references to the object
```

Never mind what the output says. The point of this example is that `help` follows *any* pointer to a function in order to retrieve its header and docstring. Therefore, it can follow the pointer to itself.

Functions that ‘look into’ functions without running them do **static analysis** of the code. Here’s an example of another static analysis function: it very crudely attempts to check if a function iterates.

```
[8]: def has_loop(function: Callable) -> bool:
    """Check if the code of `function` includes 'while' or 'for' and
    'in'."""
    text = getsource(function)
    return " while " in text or (" for " in text and " in " in text)
```

Insertion sort does a while-loop, so it has a loop.

```
[9]: has_loop(insertion_sort)  
[9]: True
```

Our key functions don't have a loop: they don't iterate over the 2-character string representing the card.

```
[10]: has_loop(suit)  # replace suit with value if you wish  
[10]: False
```

Since `has_loop` takes any function with available source code, it can be called on itself.

```
[11]: has_loop(has_loop)  
[11]: True
```

We get the wrong answer because the function just looks for the characters `while`, `for` and `in` in the source code, without distinguishing their occurrence as a keyword, in a string or in a comment.

Function `has_loop` isn't meant to illustrate robust static analysis, but rather that we *can* write functions that analyse others and themselves. As the next section will show, there are limits to the power of static analysis.

Exercise 27.3.1 (optional)

Write a function that checks if the given input function has a docstring. Test the function on the other functions in this notebook and on itself.

27.4 Undecidability

Having looked at how functions can analyse other functions and themselves, let me introduce three problems about functions that are not computable. They all happen to be decision problems. Non-computable decision problems are simply called **undecidable** problems.

27.4.1 The halting problem

Our first undecidable problem is the **halting problem**: given an algorithm and a valid input for it, i.e. that satisfies the preconditions, will the execution of the algorithm terminate for that input? (Note that we're not interested in the output or whether it's correct.) In terms of Turing machines, the problem is: given the transition table and the initial tape content, will the machine eventually stop?

We'll assume the algorithm has a single input value, to make the examples concrete. In terms of Python, what we want is a static analysis function with this heading.

```
[1]: from typing import Callable

def halts(function: Callable, value: object) -> bool:
    """Return True if and only if function(value) eventually stops."""
    ↵
    # do some highly sophisticated static analysis here
```

Note that `halts` must do static analysis: it cannot execute `function` because if the latter enters an infinite loop for `value`, so would `halts`. We would never get the desired `False` output to know that `function` doesn't halt on `value`.

It's possible to statically analyse *some* functions and determine whether they enter an infinite loop. For example, we could check if a function has no while- or for-loops, no recursive calls and only uses operations of which we know they halt. In such cases, the function will stop for every input `value` and `halts` would return `True`.

While such restricted forms on static analysis are possible, `halts` doesn't exist: nobody will ever be able to write a general algorithm that can decide whether *any* given algorithm halts on *any* given input.

Let's assume that function `halts` did exist. Then we could write functions that call it, like this one:

```
[2]: def opposite(f: Callable) -> bool:
    """Return True if and only if f(f) doesn't halt.

    Preconditions: f takes a function as argument
    """
    if halts(f, f):  # does f(f) halt?
        while True:
            pass
        return False
    else:
        return True
```

Function `opposite` uses `halts` to check if the call `f(f)` eventually stops. If it does, `opposite` gets into an infinite loop and never returns `False`. If `f(f)` doesn't halt, then `opposite` does, returning `True`. In summary, `opposite(f)` halts if and only if `f(f)` doesn't.

For example, `opposite(help)` doesn't halt because `help(help)` does: the `help` function prints its own docstring and stops, as shown in the previous section.

Now comes the sting in the tail. What would happen if we call `opposite` on itself? How will `opposite(opposite)` behave? As I just mentioned,

`opposite(f)` halts if and only if `f(f)` doesn't.

Replacing `f` with `opposite`, we have that

`opposite(opposite)` halts if and only if `opposite(opposite)` doesn't.

In other words, we have a function call that stops and doesn't stop at the same time: an impossible behaviour. This means that function `halts` can't exist: otherwise `opposite` would be able to call it and we'd get into this paradox.



Info: [Scooping the loop sniffer](#) is an entertaining and rhyming rendition of the above proof. This kind of proof is called a proof by contradiction: we assume the opposite of what we want to prove (here we assume `halts` *does* exist) and obtain a contradiction, thereby showing that our assumption was wrong.

We're asking function `halts` to predict how `opposite` (`opposite`) will behave and once we know what the prediction is, we do the opposite. So it's not possible to make a correct prediction for function `opposite` with input `opposite`. And if there's no algorithm that can solve the halting problem for one particular function and input, then there's no algorithm that can solve the problem for *any* function and input, so the halting problem is undecidable.

27.4.2 The totality problem

The halting problem asks if a given algorithm stops on *one* given input. Unsurprisingly, the more general **totality problem** (does a given algorithm stop for *all* its valid inputs?) is also undecidable.

The undecidability of the totality problem has great practical consequences. No matter how sophisticated our programming tools will ever be, they will never be able to determine for *any* program whether it will enter an infinite loop for some input. They might be able to tell us the answer for *some* programs with particular characteristics, but not for all. Since there's no systematic way to detect infinite loops, the practical consequence is that sometimes apps still 'freeze', even if they were well tested, because tests can't cover all possible inputs.

27.4.3 Rice's theorem

At this point you may be thinking that the undecidability of the totality problem is not a big deal. Only a few programs may get into infinite loops. Most programs have other kinds of errors: division by zero, indices off by one, etc. Unfortunately, detecting any of those issues is an undecidable problem too.

In fact, **Rice's theorem** states that *all* non-trivial decision problems about the behaviour of programs are undecidable. 'Non-trivial' means the decision can't be the same for all programs. For example, the decision problem 'Does the program execute zero or more steps?' is trivial: the answer is always 'yes' and hence the problem is decidable.

Note that problems about the structure of programs are decidable. For example, the decision problem 'Does the program contain a for- or while-loop?' is about the program's syntax, not its behaviour, and hence it's decidable.



Info: The theorem is named after Henry Rice, who proved it in his 1951 PhD thesis.

The practical consequence of Rice's theorem is that most software systems have errors, even when developed by very smart people following best practices, because there's no magic static analysis wand that can determine whether a program is correct. And no advance in AI or quantum computing will turn non-computable problems into computable ones.

27.4.4 The equivalence problem

The halting and totality problems have a single algorithm as input, but we can also define problems on two or more algorithms. The most famous is the **equivalence problem**: given two different algorithms for the same problem, do they compute the same output for the same valid input?

An algorithm that solves the equivalence problem would be of great value to the M269 tutors and all programming teachers. We would write *one* model solution for each programming assessment and the equivalence problem algorithm would decide if your submission is equivalent to the model solution and thus correct or not. (Assuming we write correct model answers, which we tend to do.) If the equivalence problem algorithm returns false, your tutor would still have to figure out, through tests or reasoning, for which inputs your and the model programs diverge.

More importantly, having an algorithm for the equivalence problem would allow programmers to check successive versions of their programs to make sure that their changes to the efficiency or structure of the code don't break the functionality.

Unfortunately, that algorithm doesn't exist: the equivalence problem is also undecidable.

Like the halting and totality problems, the equivalence problem can be solved for some particular programs. For example, it's possible to write a Python function that checks if two Python functions are the same, except for differences in the docstring, comments, spaces and the names of variables. However, it's impossible to write an algorithm that solves the equivalence problem for *any* two programs.

27.4.5 Reduction and computability

You learned in Chapter 26 how polynomial-time reduction can be used to classify problems as tractable or NP-hard. Here you'll see how it can be used to classify problems as computable or not.

If problem A can be reduced to problem B, then we have the following algorithm for A:

1. Transform the inputs of A into those of B.
2. Compute the output of B using any algorithm for B.
3. Transform the output of B into the one for A.

Steps 1 and 3 do not have to take polynomial time for A to be solvable: what matters is that there must be some algorithm for B we can use. In summary, a reduction of A to B tells us that if B can be solved, so can A.



Note: If problem A reduces to a computable problem B, then A is computable too.

What if we're told that A reduces to B, i.e. steps 1 and 3 are possible, but that A can't be solved, i.e. there's no algorithm for A? In that case, step 2 isn't possible: there's no algorithm for B. If there were one, the three steps would form an algorithm for A, which we're told is not possible. In summary, a reduction of A to B tells us that if A can't be solved, neither can B.



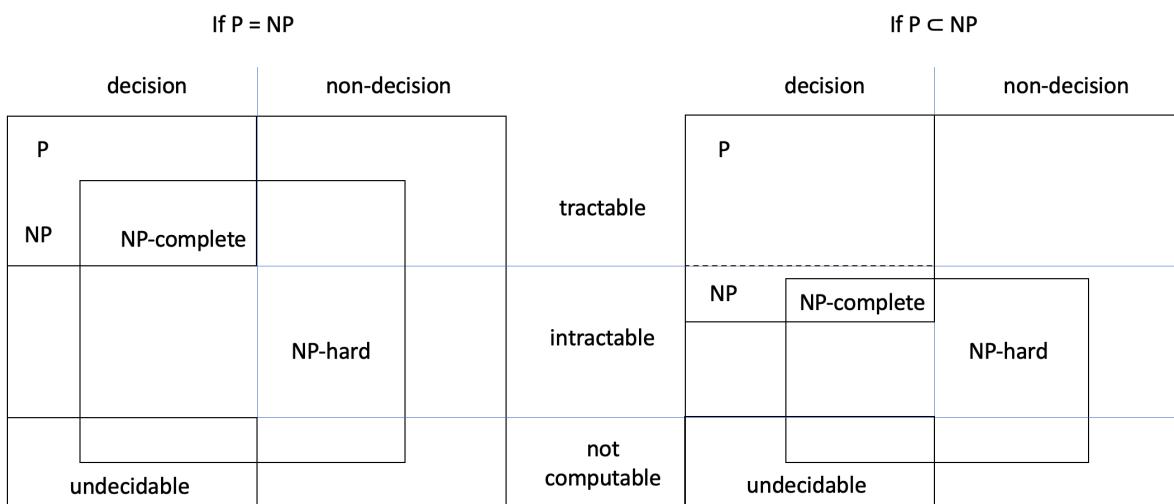
Note: If a non-computable problem A reduces to problem B, then B isn't computable either.

This fact allowed computer scientists to reduce the halting problem to the totality, equivalence and other problems to prove they are undecidable too.

27.4.6 The problem landscape

The *diagram* in Chapter 26 covers all computable problems, because each one is either tractable or intractable and is either a decision problem or not. If we add non-computable problems to that diagram, we get:

Figure 27.4.1



The new diagram shows that the undecidable problems are the intersection of the non-computable and the decision problems.

The new diagram also shows that the NP-hard class also includes non-computable problems. For example, it includes the halting problem.

As *explained before*, if an NP-hard problem A reduces in polynomial time to problem B, then B is NP-hard too. It has been shown that there's a polynomial-time reduction of the SAT problem, which is NP-hard, to the halting problem, which is therefore NP-hard too.

Exercise 27.4.1

Knowing that the halting problem reduces in polynomial time to the totality and to the equivalence problems, does it mean that these two problems are NP-hard too? Explain why or why not.

Hint Answer

Exercise 27.4.2

Bob is confused.

‘We were told that SAT reduces to the halting problem,’ he tells Alice.

‘Indeed,’ replies Alice.

‘We were also told that if a non-computable problem reduces to some problem B, then B isn’t computable either,’ continues Bob.

‘Yes,’ confirms Alice.

‘So shouldn’t SAT be undecidable, like the halting problem?’ asks Bob.

How can Alice explain to Bob what’s wrong with his reasoning?

Hint Answer

27.5 Summary

This chapter introduced a formal model of computation, the Turing machine, and showed the theoretical and practical limitations of computation, by exhibiting three non-computable problems.

27.5.1 Turing machines

A **Turing machine** consists of

- an infinite **tape** made of **cells**, each having one **symbol**
- a read/write **head** that is always over a single cell
- a non-empty set of **states** (including an **initial state**)
- a non-empty set of symbols (including the **blank** symbol)
- a **transition table** that defines the algorithm followed by the machine.

The blank symbol represents an ‘empty’ cell.

The transition table has at most one entry per state–symbol pair, indicating one **execution step**:

- the symbol that is written (it may be the symbol that was read)
- how the head moves (one cell to the left or right or not at all)
- the next state (it may be the same as the current state).

A machine stops if there's no entry in the transition table for the current state and the current symbol under the head.

In M269, the tape has a start but no end, the initial state is always called 'start' and the head is initially over the first, left-most cell. The machine stops with an error if the head moves to the left of the start position.

The **input** (respectively, **output**) of the machine are the symbols from the initial (respectively, final) position of the head onwards, until the start of the infinite sequence of blanks.

A **configuration** is given by the position of the head, the content of the tape and the state. In the initial configuration, the head is in the left-most position, the tape contains the machine's input followed by infinite blanks and the state is 'start'. The execution of the machine can be seen as a sequence of configurations, starting with the initial configuration. Each transition from one configuration to the next corresponds to the execution of one step.

Turing machines are a formal model of computation: they define what algorithm, complexity and computability mean.

File `m269_tm.py` provides function `run_tm(machine, input, debug)` that executes a given machine on a given input and returns the output. Input and output are represented as lists, with `None` for the blank symbol. The Boolean `debug` parameter is false when omitted; if it's given and true, the configurations are printed as the machine executes.

A machine is defined by its transition table, represented with a Python dictionary that has state–symbol pairs as keys and symbol–movement–state triples as values.

File `m269_tm.py` includes two further functions `check_tm(machine, in, out)` and `check_tm_tests(tests, in, out)` to check transition table `machine` and test table `tests` with input symbols `in` and outputs symbols `out`. Passing the checks doesn't guarantee that `machine` and `tests` are correct.

The same file also defines function `test_tm(machine, tests, show_tests)` to test a machine against a test table. The Boolean `show_tests` parameter is false when omitted; if it's given and true, the name of each test is printed before it's executed.

27.5.2 Computability

The **Church–Turing thesis** states that anything that can be computed can be computed by a Turing machine. The thesis suggests that anything people agree to call an algorithm can be written as a transition table for a Turing machine. It has been proven that the **lambda calculus** (a computational model based on functions) and the various definitions of Turing machine are all equivalent, giving strength to the thesis.

A computational problem is **computable** if there's an algorithm, e.g. in the form of a Turing machine transition table or of a Python program, that solves the problem. An **undecidable** problem is a non-computable decision problem. The following problems are undecidable.

- **Halting problem:** given an algorithm and a valid input for it, i.e. that satisfies the problem's preconditions, does the algorithm eventually stop for that input?
- **Totality problem:** given an algorithm, does it stop for all valid inputs?

- **Equivalence problem:** given two algorithms for the same problem, do they produce the same output for each valid input?

The undecidability of these problems has practical implications: it's impossible to determine if a program will get into an infinite loop, and it's impossible to determine if modifications to a program, e.g. to make it simpler or more efficient, will maintain its original behaviour. The problems can be solved with **static analysis** (the ability to analyse programs without running them) for particular kinds of programs or modifications, but not for *any* program or modification.

Python supports static analysis with function `getsource` in module `inspect`.

Rice's theorem states that all non-trivial decision problems about the behaviour of an algorithm are undecidable. A decision problem is trivial if the output is the same for all inputs.

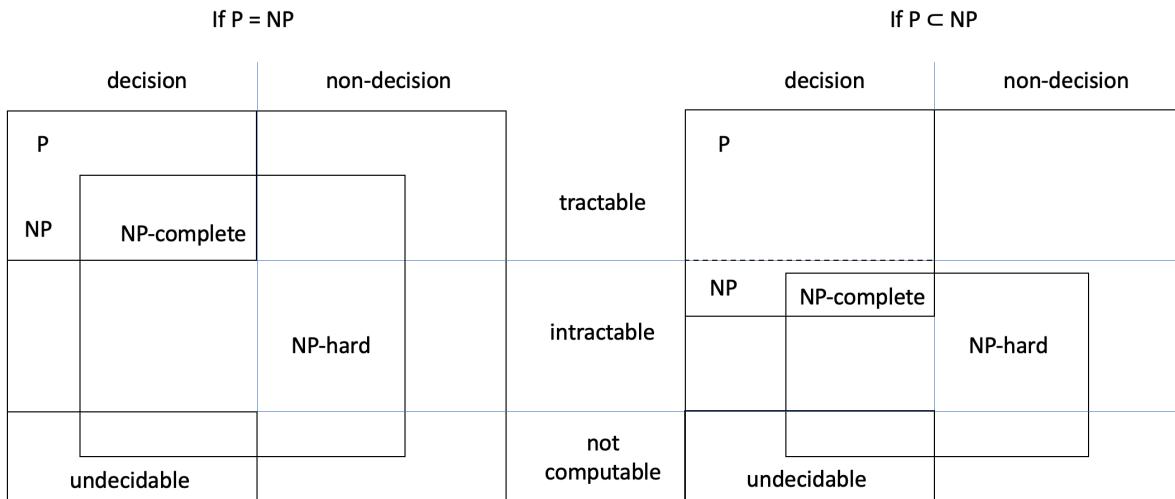
If problem A reduces (not necessarily in polynomial time) to problem B, then:

- if B is computable, so is A
- if A isn't computable, neither is B.

The halting problem can be reduced (in polynomial time, as it happens) to the totality and to the equivalence problems, thus proving their undecidability.

The classes of problems, now including non-computable problems, are as follows.

Figure 27.5.1



SAT reduces in polynomial time to the halting problem. This entails that the halting, totality and equivalence problems are NP-hard.

CHAPTER 28

TMA 03 PART 2

You have now reached the end of M269. Well done!

The remaining time is for you to complete and submit the final TMA, which follows different rules from the previous TMAs: see the ‘[Assessment](#)’ tab of the M269 website.

This chapter provides some advice that complements the summaries of Chapters [26 \(complexity classes\)](#) and [27 \(computability\)](#). The guidance in Chapters [5](#), [10](#) (in particular about checking your TMA), [15](#), [20](#) and [25](#) still applies.

Before starting to work on this chapter, check the M269 [news](#) and [errata](#).

28.1 Problem classes

This section summarises how you can prove that a problem *Given* is in a particular class. Some proofs are based on the following facts:

If ...	Then <i>Given</i> is ...
<i>Given</i> reduces to a computable problem	computable
<i>Given</i> reduces in polynomial time to a tractable problem	tractable
an NP-hard problem reduces in polynomial time to <i>Given</i>	NP-hard

Note that to determine whether *Given* is computable, we don’t care about the complexity of the reduction. A problem is (or isn’t) computable if there is (or isn’t) *some* algorithm that solves it: the algorithm’s complexity is irrelevant.

Note also that the reduction goes in one direction when showing that a problem is computable or tractable, and goes in the opposite direction when showing that the problem is NP-hard.

28.1.1 Computable

There are two ways of proving that *Given* is computable. The direct way is:

- Provide an algorithm that solves *Given*.

What ‘provide’ means depends on what you’re asked for: it may be just an outline or a Python implementation or a Turing machine. Do *not* analyse the complexity of your algorithm (unless you’re asked to, of course) because its efficiency is irrelevant. Go for the simplest algorithm you can think of, e.g. a brute-force search.

The second, indirect, way uses reduction as an algorithmic technique to provide an algorithm for *Given* (first row of the above table):

1. Select a problem *Chosen*: usually it’s a more general problem than *Given*.
2. Explain why *Chosen* is computable: either provide an algorithm for *Chosen* or point to such an algorithm in this book, by referring to a section or exercise.
3. Show that *Given* reduces to *Chosen* by providing the input/output transformations.

Since the efficiency of the algorithm for *Given* is irrelevant, the input/output transformations do *not* have to be polynomial.

The second way of proving computability may be simpler than the first way, if the given problem *Given* is similar to or a special case of a problem solved in this book, and the transformations are simple.

Unless asked otherwise, you can choose either way of proving computability and you don’t need to write algorithms in detail: outlines will do.

28.1.2 Tractable

To prove that *Given* is tractable, you must prove that it’s computable by an algorithm with worst-case polynomial complexity. So it’s a computability proof with an additional complexity analysis step. The direct way is:

1. Provide an algorithm that solves *Given*.
2. Analyse the worst-case complexity of the algorithm to confirm it’s polynomial.

The confirmation can be as simple as stating, for example, ‘The worst-case complexity is quadratic and therefore polynomial.’

The second, indirect, way follows the second row in the above table:

1. Select a problem *Chosen*: usually it’s a more general problem than *Given*.
2. Explain why *Chosen* is tractable: either provide a polynomial algorithm for *Chosen* or refer to the section or exercise in this book that has such an algorithm.
3. Show that *Given* reduces to *Chosen* by providing the input/output transformations.
4. Analyse the worst-case complexities of the transformations, confirming they are polynomial.

28.1.3 Intractable

To prove that *Given* isn't tractable:

- Show that the size of the output is non-polynomial in the size of the input.

Typically, intractable problems ask for all (or most of the) subsets or permutations of the input. Even if each subset or permutation could be produced in constant time, producing the whole output would take exponential or factorial time in the worst case.

28.1.4 NP-hard

To prove that *Given* is NP-hard, use the third row of the table:

1. Choose an NP-hard problem *Chosen* from [Section 26.6.3](#).
2. Show that *Chosen* reduces to *Given* by providing the input/output transformations.
3. Analyse the worst-case complexities of the transformations, confirming they are polynomial.

Steps 1 and 2 are the creative part of the proof: there's no recipe for them. However, here are some general suggestions that might (or might not) work for the problem you're given.

If problem <i>Given</i> is about ...	Then problem <i>Chosen</i> might be ...
putting items in some order	(decision) TSP or longest path
numeric values	subset sum or 0/1 knapsack
selecting items that are in a binary relation	maximal independent set
satisfying constraints	SAT

28.1.5 NP

To prove that *Given* (which must be a decision problem) is in NP:

1. Define a certificate for each input that leads to a 'yes' decision.
2. Provide the verifier's algorithm, explaining why it does confirm 'yes' decisions.
3. Analyse the verifier's worst-case complexity, confirming it is polynomial.

In step 1, you must think of what piece of information would convince you that the input to problem *Given* leads to a true output. Try to rephrase the decision problem as a search problem: 'Is there an X that satisfies condition C?' or similar. If the answer is 'yes', then there is such an X, and that's the certificate.

For example, the SAT problem asks 'Is there an interpretation that makes the input expression true?' and so the certificate is one such interpretation. Another example: the decision TSP asks 'Is there a tour of the input graph with total weight w or less?' and so the certificate is one such tour.

In step 2, you must provide an algorithm that checks the certificate against the input to confirm the decision is 'yes'. To know what the verifier must do, it often helps to look at the postconditions of *Given*.

For example, the postcondition of SAT is: the output is true if and only if there's an interpretation that makes the Boolean expression true. The interpretation is the certificate and the verifier checks it indeed makes the input expression true. As for the decision TSP, its postcondition is: the output is true if and only if there's a tour of the input graph with total weight equal to or less than the input integer. Hence, the certificate is a tour and the verifier checks its weight is indeed not larger than the given integer.

28.1.6 NP-complete

To prove that *Given* is NP-complete, just follow the definition of NP-completeness:

1. Prove that *Given* is in NP.
2. Prove that *Given* is NP-hard.

28.2 Turing machines

If you're asked to write a Turing machine to solve a problem, I suggest you start working through a simple and short example with pencil and paper, to figure out how the head moves and which symbols have to be written.

When you need some extra piece of information to know how to process the current symbol, think whether that information is best conveyed through an additional state or by writing additional symbols on the tape. If you need to mark a position of the tape so that you can later return the head to it, you will likely need a new symbol to mark the position and a new state for when the head is moving towards that position.

If your algorithm has various ‘phases’, then you will likely need at least one state per phase. Examples of phases are: processing the input left to right, writing the output, and returning the head to the start of the output.

Read the problem description carefully, especially regarding the expected output. The output is the symbol sequence starting from the head's final position. Returning the head to the original left-most position may be the wrong thing to do for the problem at hand.

When writing the transition table as a Python dictionary, you can write the individual transitions in any order, but the two best orders are to either group the transitions by state or to list the transitions in the order they are executed by the algorithm, as illustrated in the *parity bit* example.

You may wish to add some comments to your transition table to help your tutor follow your reasoning.

If you're asked for an outline or explanation of your algorithm, do *not* simply translate the transition table to English, row by row. Instead, give a high-level view of the algorithm. Explain what each state is for, what is done in each state, and how and where the machine stops.

28.3 Python

Unless a TMA 03 question explicitly states otherwise, your code can only use the types, classes, methods, functions, statements, constants and code templates introduced in this book. Here's a non-comprehensive index of them, in addition to Sections [10.4](#) and [20.3](#).

28.3.1 Standard library

- function `getsource` from module `inspect` ([27.3.1](#))

28.3.2 M269 library

Undirected graphs (file `m269_ungraph.py`):

- function `connected_components` ([21.1.3](#))

Directed graphs (file `m269_digraph.py`):

- functions `reverse` and `strongly_connected_components` ([21.2.3](#))
- function `topological_sort` ([21.3.2](#))

Turing machines (file `m269_tm.py`):

- functions `check_tm`, `check_tm_tests`, `test_tm` and `run_tm` ([27.1.3](#))

28.4 Final words

I hope you found M269 interesting and useful. No matter what computational problems you will face, it's likely that you can tackle them efficiently with some of the data structures (linked lists, hash tables, trees, graphs, etc.) and techniques (divide-and-conquer, greed, backtracking, dynamic programming) you learned.

M269 has given you a process that you should continue to apply, even for simple problems that don't require sophisticated data structures or algorithms: define the problem, write tests, design an algorithm, analyse its complexity, implement it if it's efficient enough, test it, and confirm its efficiency with run-time performance measures. Remember that the process is iterative: never be content with your first attempt.

Please keep writing elegant, documented, tested, readable code: those who have to maintain your code will thank you.

Last but not least, I hope M269 inspired you to learn more about this fascinating field. We haven't covered specialised algorithms and data structures for text, sound and image processing, bioinformatics, databases or whatever domain you're interested in. We also haven't learned about

- memory complexity to better assess how efficient a solution is
- algorithmic techniques for handling NP-hard problems
- algorithm engineering techniques to improve run-time performance
- parallel and distributed algorithms

and much more, but what you learned in M269 should help you start exploring those topics. On behalf of the whole module team, best wishes for your lifelong learning!

CHAPTER 29

HINTS

This chapter contains all hints for exercises throughout the book. Not every exercise has a hint.

29.1 Numbers and sequence

29.1.1 Expressions

Exercise 2.3.1 hint

If a result is correct, it doesn't mean it was obtained in the right way.

29.1.2 Functions in mathematics

Exercise 2.5.1 hint

You may wish to remind yourself of the *first exercise* in the previous section, and its solution, which provides the formula for the postcondition in this exercise.

You must also consider how to represent the VAT rate, e.g. as a percentage, and what unit the prices are in.

29.1.3 Functions in Python

Exercise 2.6.1 hint

Remember to separate parameters with commas and to add the output type to the header. If the conditions are too long for a single line, write them as a list: see the start of the section.

29.1.4 Complexity

Exercise 2.7.1 hint

How many operations does the algorithm execute and what is the complexity of each one?

29.1.5 Run-times

Exercise 2.8.1 hint

What are the values of 1^y for ever growing exponents y ? What could be the impact on the run-time? Would we be able to ascertain whether exponentiation is $\Theta(y)$?

Exercise 2.8.2 hint

Copy and paste the code cell that measures the run-time of addition. Remove the measurements for values below 256. Use the find and replace command to change the operation, the number of runs and the number of loops. For a complete analysis, you should double the digits and double the values, but if you're pressed for time, only do one experiment.

29.2 Booleans and selection

29.2.1 Booleans

Exercise 3.1.1 hint

Read again the definitions of conjunction and disjunction at the start of the section, and then adapt the conjunction template accordingly.

Exercise 3.1.2 hint

There are only two Boolean values, so what's the relation between the input and the output?

Exercise 3.1.5 hint

A conjunction is false if either operand is false. The left operand is true. How can you make the right operand of the conjunction become false?

Exercise 3.1.6 hint

As we saw in the previous chapter, names are case-sensitive in Python, e.g. `false`, `False` and `FALSE` are different names.

Exercise 3.1.7 hint

First insert parentheses according to the rules, then evaluate them in order, from most to least nested and left to right.

29.2.2 Decision problems

Exercise 3.2.1 hint

Each input variable is true in half of the eight instances and false in the other half. So which instances are missing and what are their outputs?

Exercise 3.2.3 hint

What must the values of `in_flight_mode` and `wifi_on` be for only the disjunction to short-circuit?

29.2.3 Boolean expressions**Exercise 3.3.2 hint**

There are three errors. To simplify the expression, see if part of it is redundant.

29.2.4 Classification problems**Exercise 3.4.1 hint**

You must reverse the order in which the grade boundaries are checked, from highest to lowest.

Exercise 3.4.2 hint

Does it classify each mark into the correct grade?

Exercise 3.4.3 hint

Does it produce all possible grades?

29.2.5 Practice**Exercise 3.5.1 hint**

1. Can the problem be formulated as a yes/no question?
2. Write a postcondition of the form ‘output = value if and only if conditions on inputs’.

Exercise 3.5.2 hint

There are two inputs, so you must add a column. A decision problem is a particular case of a classification problem, so make sure you have at least one test for each possible output. Think also of the edges cases: what are the boundary values?

Exercise 3.5.3 hint

It’s a single assignment.

Exercise 3.5.4 hint

State the input and output types in the header. Write a docstring with the pre- and postconditions. Compare the value returned by each function call against the expected output.

Exercise 3.5.5 hint

The inputs can be any real numbers. You can't use the max function from [Section 2.2](#) in the postconditions, because we're assuming it doesn't exist.

Exercise 3.5.8 hint

1. What's the first year for which you can say whether it's a leap year or not? If you can't write the postconditions as one 'if and only if' statement, then list several mutually exclusive and comprehensive conditions, depending on whether the year is divisible by 4, 100 or 400. Remember which operation checks for divisibility.
2. You need at least two more cases: the year isn't divisible by four; the year is divisible by four but not by 100.

Exercise 3.5.9 hint

First write an algorithm that closely follows your postconditions and then simplify it.

29.3 Sequences and iteration

29.3.1 The Sequence ADT

Exercise 4.1.1 hint

Can the precondition be satisfied for the empty sequence?

29.3.2 Strings

Exercise 4.2.1 hint

Use the membership operation.

Exercise 4.2.2 hint

All expressions are incorrect but one.

Exercise 4.2.3 hint

In which cases are the fewest or most characters copied to the output? Look at the given code examples.

Exercise 4.2.4 hint

What should the output be if the input string is empty? For each input, execute the algorithm 'mentally'. What's the value of `middle`? Why doesn't computing the first or second half raise an index error?

Exercise 4.2.5 hint

Think of the addition operation: in general it is linear in the largest of the sizes of the integers being added, but in M269 we assume it takes constant time.

Exercise 4.2.6 hint

There are two solutions I can think of. Both use concatenation and repeated concatenation. One solution creates the exact number of ellipses needed. The other solution creates one extra ellipsis and then slices it off.

29.3.3 Iteration

Exercise 4.3.1 hint

What are the first and last indices? Change the loop to ‘for *index* from ... down to’ and translate it to `for index in range(start, end-1, -1)`.

29.3.4 Linear search

Exercise 4.4.1 hint

Remember that the smallest possible input is an edge case and should be included.

Exercise 4.4.2 hint

Mentally ‘execute’ the algorithm for a small input from your test table. You only need to find one counter-example to prove that an algorithm is incorrect.

Exercise 4.4.4 hint

When does such an algorithm do the fewest iterations? When does it do the most? Since a scenario is a group of problem instances, describe their form.

29.3.5 Tuples and tables

Exercise 4.5.1 hint

In which row and column is the sought data?

Exercise 4.5.2 hint

Do you need to iterate over the indices or can you iterate over the cells?

Exercise 4.5.3 hint

The most important information to know is where the ladder bottoms and snake heads are, and to where a pawn landing on those positions must move.

If the board is represented as a table, how is the position of each pawn represented and is it easy to compute a pawn’s new position?

29.3.6 Lists

Exercise 4.6.1 hint

The input and output become a single input/output variable. You must use the pre- x and post- x notation to distinguish the items as they are before and after the reversal.

Exercise 4.6.2 hint

The complexity of the append operation is given earlier in this section. The complexity of concatenation is in [Section 4.1](#).

Exercise 4.6.3 hint

Is it easier to first add a column and then the game or vice versa?

29.3.7 Reversal

Exercise 4.7.1 hint

The algorithm has to add items from the original to the reverse list. The most efficient way to add items is to append them. How must you go through the original list, so that appending items reverses their order? If it helps, do as I did: choose a problem instance and go through it with your left and right index fingers.

Exercise 4.7.3 hint

See [Section 4.3](#) for how to iterate backwards in Python. You can use the same test table, as it's the same problem.

Exercise 4.7.4 hint

Which of the algorithms for lists is easiest to modify for strings: the one using the insertion operation or the one using the append operation?

Exercise 4.7.5 hint

The analysis is like that for the original algorithm for lists, using insertion.

Exercise 4.7.6 hint

We must swap the first item with the last item, the second with the penultimate, etc. Start with your left index finger on the first item and your right index finger on the last item. Swap the items pointed to by your fingers. Move your left finger forward (to the right) and your right finger backward (to the left). Swap those two items. When do you stop swapping? Make sure your algorithm works for lists of length 0 and 1.

29.3.8 Optional practice

Exercise 4.8.1 hint

This decision problem can be seen as a search problem. The algorithm is a linear search that stops as soon as a character other than the four letters is found. Make sure your algorithm returns false for the empty string.

Exercise 4.8.2 hint

Alice's approach requires a variable to remember what is the smallest value found so far during the search. Use the generic test function with the same test table for both approaches.



Note: A shorter algorithm (with fewer instructions) isn't necessarily more efficient than a longer algorithm.

Exercise 4.8.3 hint

Test your algorithm with strings of the same and different lengths, and when one operand is the empty string.

Exercise 4.8.4 hint

There's a one-line algorithm that uses the reversal function on strings (not the in-place operation) to check whether the reversed string is equal to the input string. This algorithm wastes memory by using a new but temporary list.

There's a better algorithm that is similar to the in-place reversal. Test it with the empty string and strings of odd and even lengths.

Exercise 4.8.5 hint

In a string, all items (characters) are pairwise comparable according to the Unicode standard. Once the characters are sorted, you can iterate over them, count how many consecutive characters are the same, and keep the highest count so far.

29.4 Implementing sequences

29.4.1 Developing data types

Exercise 6.3.1 hint

Use the `length` and `get_item` methods to do a linear search that stops as soon as the item is found.

29.4.2 Bounded sequences

Exercise 6.4.1 hint

The ‘ready, set, go’ example works, even though the append operation is implemented in the abstract class in terms of the insert operation. Why does inserting at the end work, but not inserting at the start?

Remember that the `test_insert_start` *function* always inserts at index 0 and so inserts the numbers in inverse order. For a length of 3, the test first inserts 2, then 1 and finally 0. It then checks whether the resulting sequence is 0, 1, 2.

Exercise 6.4.3 hint

The algorithm to remove an item is symmetric to the insertion algorithm: instead of going backwards from the end to the index, it goes from the index to the end; instead of copying each item to the right, it copies to the left.

29.4.3 Using dynamic arrays

Exercise 6.6.1 hint

An item is removed from a dynamic array in the same way as from a static array, so you can copy your *previous code*.

One possible shrinking policy is: if the new sequence length, after removing the item, is 50% of the capacity, set the new capacity to 75% of the current one. For example, if the length is 50 and the capacity is 100, the new capacity will be 75, which allows to add 25 items before the capacity is increased.

29.4.4 Linked lists

Exercise 6.7.1 hint

The algorithm is similar to insertion but simpler because no node is created. Like for insertion, you need a reference to the node before and you need to handle the case of removing the first (and possibly only) item.

Exercise 6.7.3 hint

It’s another space–time tradeoff…

29.5 Stacks and queues

29.5.1 Stacks

Exercise 7.1.1 hint

The push operation is effectively the append operation on sequences.

Exercise 7.1.2 hint

Pop and peek remove and access the item at index $|values| - 1$.

Exercise 7.1.3 hint

The answer to this sort of question is almost always ‘it depends’. Consider both the memory usage and run-time of both approaches.

29.5.2 Using stacks

Exercise 7.2.1 hint

Which scenarios lead to the least or most work done by the algorithm? Remember that scenarios are about the form of the input, not their size. The empty string (input of size zero), or any other single input value, isn’t a scenario by itself.

Exercise 7.2.3 hint

Look again at the balanced brackets algorithm. If you remove all parts that handle square brackets, do you still need a stack?

Exercise 7.2.4 hint

Look for a LIFO relation, i.e. where the last of something is processed first. That something is what you need to keep in a stack.

Exercise 7.2.5 hint

First write a table like this to go through an example:

Stack	Remaining expression	Action

When and what do you push on or pop from the stack?

29.5.3 Queues

Exercise 7.3.1 hint

Which one is the most appropriate to remove the tasks as they’re completed?

Exercise 7.3.3 hint

Think about how one would dequeue the front item, in the last node.

Exercise 7.3.4 hint

What are the elements of the queue? What is the front item, what is the back item? How can a circular arrangement be represented by a linear arrangement?

Exercise 7.3.5 hint

The rhyme has 28 syllables. The algorithm simulates $n - 1$ rounds of counting, at the end of which only one child remains. How can the algorithm use the queue to simulate Alice pointing to the next child and one child leaving the circle each round?

Exercise 7.3.6 hint

You need a nested loop: the outer loop iterates once per round; the inner loop iterates over the 27 syllables.

Exercise 7.3.7 hint

Remember that all `Queue` operations take constant time and that a fixed number of constant-time operations takes constant time, no matter how high that number is.

29.5.4 Priority queues

Exercise 7.4.1 hint

What are the best- and worst-case complexities of sorting?

Exercise 7.4.2 hint

Remember that the algorithm must keep items of the same priority in FIFO order. If existing items of the same priority as the new item are, say, at indices 3 to 6 (inclusive), where should the new item be inserted?

Exercise 7.4.4 hint

Trick question: what are the best- and worst-case complexities of inserting?

Exercise 7.4.5 hint

Lists are implemented with dynamic arrays.

Exercise 7.4.7 hint

Look at the reminders example code.

Exercise 7.4.8 hint

It's a minimal change in a single method.

29.6 Unordered collections

29.6.1 Maps

Exercise 8.1.2 hint

Remember that maps are unordered collections (keys aren't in any particular order) and that you can iterate over the keys.

Exercise 8.1.4 hint

For the first question, remember how maps can be implemented.

For both questions, consider situations like boarding a plane: there are over 100 people boarding but only a few priority categories, like wheelchair users, families with small children, first-class and economy passengers.

29.6.2 Dictionaries

Exercise 8.2.2 hint

Think about the algorithm incrementally, for each entry in the original dictionary. Design an algorithm that works for the first entry ('alface'). Check if it can also handle a map with the first two entries ('alface' and 'carro'). If it can't, change the algorithm. Then consider the third entry, and so on.

As you progress, manually fill the inverted map table while thinking (or even speaking aloud) what the algorithm must do.

29.6.3 Hash tables

Exercise 8.3.1 hint

Look at the example hash table and imagine the slots only have values.

Exercise 8.3.2 hint

Look in the class for where the keys are used and how.

Exercise 8.3.3 hint

What does the load factor influence?

29.6.4 Sets

Exercise 8.4.1 hint

Every number is either even or odd.

Exercise 8.4.2 hint

If you have to quickly compute the intersection of two sets, one with 1000 items and the other with only three, how would you proceed?

Exercise 8.4.3 hint

The expression has the same best- and worst-case complexities: it's always linear in the smallest of the two sets. If the two sets aren't disjoint, can you find that out more quickly?

Exercise 8.4.4 hint

As you iterate through the ranking, extract the school's name from the team's name and check if that school already got a certificate.

As for the tests, what are the edge cases?

29.7 Practice 1

29.7.1 Pangram

Exercise 9.1.1 hint

Think of the smallest and largest inputs and outputs.

Exercise 9.1.2 hint

Does the order of the characters in the text matter? Can you directly compute the missing letters?

Exercise 9.1.3 hint

An operation takes constant time if the input size has a fixed bound.

Exercise 9.1.4 hint

Two algorithms may have the same complexity but one may execute a far greater number of operations than the other.

Exercise 9.1.6 hint

Steps 1–4 of the third algorithm can be implemented with a single Python line: `ordered = sorted(set(LETTERS) – set(text))`.



Info: These exercises are based on problem [Quick Brown Fox](#).

29.7.2 Election

Exercise 9.2.1 hint

Look at the current tests and think of other ways in which the votes may be distributed among candidates.

Exercise 9.2.2 hint

What ADT is best for recording how many votes each candidate got?

Exercise 9.2.3 hint

Think of the relation between the number of votes and the number of candidates.

Exercise 9.2.4 hint

Two approaches may be equivalent in one case (best or worst) but one may be better in the other case. The complexity of sorting is in the *Chapter 4 summary*.

Exercise 9.2.5 hint

Look at the summary of the *dictionary operations* if you need a reminder on how to use them.

29.7.3 Voucher

Exercise 9.3.1 hint

The algorithm should go only once through the store to find all ‘matching’ products, i.e. that add up to the voucher’s amount. Thinking of concrete examples always helps: point with a finger through one of the problem instances and think what needs to be done for each product. For example, in the fifth test case, with *voucher* = 7, as you go through the products, how can you know that F2 matches F5? What data structure can help you quickly find a matching product?

Exercise 9.3.2 hint

In a best-case scenario, how many product pairs add up to the voucher?

Exercise 9.3.3 hint

How many product pairs can at most add up to the voucher?

Exercise 9.3.4 hint

Do we know the map’s keys in advance?

29.7.4 Trains

Exercise 9.4.3 hint

Think with your hands. Draw the track on paper, take coins, dice or other small objects to represent the wagons. You need two copies of each, one initially in the east and the other in the west to indicate the desired order. Move the objects from the east around, following the example. How does the configuration in the west change to ‘tick off’ the wagons already processed? Try to solve each problem instance in the provided tests.

Which ADT(s) best represent the wagon configurations, and how they change, in the east, west, and in the station?

Exercise 9.4.4 hint

When does the algorithm not stop early?

Exercise 9.4.5 hint

What is a simple way to write Python code using queues and stacks?

29.7.5 SMS

Exercise 9.5.1 hint

Think of the least data you need, and in which form, so that you can find the words that complete a given prefix. Alternatively, take as a starting point the data structure that uses the least memory and the ADTs that can be implemented with it.

Exercise 9.5.2 hint

Is it better to have a sequence of word–score pairs or score–word pairs? Don’t forget that the `__init__` method must store the data in an instance variable which is then used by the `completions` method.

See the [Chapter 4 summary](#) for sorting.

Use `slicing` to check if a word starts with a prefix.

Exercise 9.5.3 hint

What are the prefixes that lead to the worst complexity to find the completions? How do the worst run-times grow from 100 to 10,000 words? Extrapolating, what can you expect for 100,000 words?

Exercise 9.5.4 hint

Do as much as possible in the initialisation and as little as possible in the completions operation. What ADT allows you to return the completions in constant time, no matter what prefix is given? Think backwards: first design the completions operation to know what collection the initialisation must produce.

29.8 Exhaustive search

29.8.1 Linear search, again

Exercise 11.1.1 hint

Remember the '*Eeny meeny*' example.

Exercise 11.1.3 hint

What type of collection should the *solutions* variable be? What else do you need to change to keep the order?

Exercise 11.1.4 hint

What collection type is appropriate?

Exercise 11.1.5 hint

Keep a collection of the equally best candidates found so far. Each new candidate is compared against only one of those best candidates, because they're all equally good. What happens if the current candidate is as good as one of the best? What happens if the current candidate is better?

29.8.2 Factorisation

Exercise 11.2.2 hint

What are the two factors of a prime number?

29.8.3 Constraint satisfaction

Exercise 11.3.1 hint

Supposing each iteration takes 100 ns, multiply the number of iterations by 100 and divide by 1 billion to get the time in seconds.

29.8.4 Searching permutations

Exercise 11.4.1 hint

Can you think of 'symmetric' tours that have the same cost?

29.8.5 Searching subsets

Exercise 11.5.2 hint

It's similar to the TSP, but with subsets instead of permutations and with the cost of a tour now being the value of the items in the knapsack.

29.8.6 Practice

Exercise 11.6.2 hint

What are the candidates? Are there equivalent candidates that shouldn't be generated?

Exercise 11.6.4 hint

What can you do with the input sequence if items are comparable?

Exercise 11.6.6 hint

Can you think of a similar problem?

Exercise 11.6.7 hint

It's possible to do both. Think of conditions under which an item or a whole subset doesn't have to be considered.

Exercise 11.6.8 hint

Remember the three questions of a brute-force search: What are the candidates? How are they generated, one by one? How is each candidate tested to know if it's a solution?

Exercise 11.6.9 hint

The algorithm has two inputs so the complexity should refer to $|s|$ and $|t|$.

Exercise 11.6.11 hint

Can the PIN have 0 as digit? One approach generates and tests all tuples of four digits. The other approach generates 4-combinations and permutations.

Exercise 11.6.12 hint

Does the problem have an input?

29.9 Recursion

29.9.1 Recursion on integers

Exercise 12.2.1 hint

Just follow the questions:

- What is the lowest input value and is it even or odd?
- If you know whether $n - 1$ is even or odd, what does it say about the evenness of n ?

29.9.2 Length of a sequence

Exercise 12.3.1 hint

Answer the usual two questions, adapted to this problem:

1. What is the smallest sequence and its sum?
2. If you know the sum for the tail of a sequence S, what's the sum of S?

29.9.3 Inspecting sequences

Exercise 12.4.2 hint

You need two base cases and must compare the head with the number after it. How can you access the second item of a sequence?

29.9.4 Creating sequences

Exercise 12.5.1 hint

For the tests, think of edge cases and a ‘normal’ input sequence.

Exercise 12.5.2 hint

If you have the reverse of the tail, how must you combine it with the head to get the reversed sequence?

Exercise 12.5.3 hint

Like indexing, this operation has two arguments that can be decreased: the sequence *items* and the integer *index*. What's the base case?

29.9.5 Avoiding slicing

Exercise 12.6.1 hint

The transformation is similar to what was done for the membership operation. Copy the pattern to the answer cell. Add two extra arguments to all calls of the search function. Replace all occurrences of the head and tail operations. For the base case, check if the slice to be searched is empty.

Exercise 12.6.2 hint

For the base case, how can you check the slice has length 1?

29.9.6 Multiple recursion

Exercise 12.7.1 hint

Answer the three questions. Here's a hint for each one.

1. Keep it simple, as in the maximum example.
2. If the items exists in the sequence, it must be in one of the halves.
3. For which lengths can't a sequence be divided into shorter parts?

29.10 Divide and conquer

29.10.1 Variable decrease

Exercise 13.3.1 hint

What's the range of the candidates? How should you generate them?

29.10.2 Binary search

Exercise 13.4.2 hint

What other algorithm have you seen that halves its input and only recurs into one half?

29.10.3 Binary search variants

Exercise 13.5.2 hint

Use a loop to decrease the size until it's 1 or 2. Handle the base cases after the loop.

Exercise 13.5.3 hint

What half should you search if $\text{numbers}[\text{middle}] > \text{middle}$?

29.11 Sorting

29.11.1 Insertion sort

Exercise 14.3.1 hint

When in doubt, try it out. Remove `ed` twice from the function calls and explain what happens.

29.11.2 Selection sort

Exercise 14.4.2 hint

Like for the previous version of the algorithm, there are seven lines and six swaps.

29.11.3 Merge sort

Exercise 14.5.1 hint

For the recurrence relation, account for slicing each half, for making one recursive call on each, and for merging both halves.

29.11.4 Quicksort

Exercise 14.6.1 hint

What are the lengths of the partitions?

29.11.5 Quicksort variants

Exercise 14.7.1 hint

All being equal is a special case of already being sorted.

Exercise 14.7.2 hint

The choices are: decrease by a constant amount, decrease by a constant factor, decrease by a variable amount.

Exercise 14.7.3 hint

Replace step 2.2 with an if-statement that handles the three cases mentioned: return the pivot, search partition *smaller* or search partition *larger*.

In addition, you must handle the base case of a short sequence that can't be split. In two-way quicksort, that's when the sequence is empty or has a single item. What's the corresponding base case for quickselect and how is it handled?

29.12 Rooted trees

29.12.1 Algorithms on trees

Exercise 16.2.1 hint

If you know the height of each subtree, what's the height of the whole tree? As usual, working out the result by hand for a problem instance helps. For the first tree in this chapter, the left subtree (with the multiplication) has height 3 and the right subtree, with leaf 6, has height 1. From an earlier exercise you know the height of the whole tree is 4. Can you derive a general expression based on the heights of the left and right subtrees?

29.12.2 Traversals

Exercise 16.3.1 hint

Add a second base case.

29.12.3 Binary search trees

Exercise 16.4.1 hint

What is different from the membership operation?

Exercise 16.4.2 hint

Leverage the ordering property of BSTs.

Exercise 16.4.3 hint

The typical base case is the empty tree. Does it make sense for this operation? If not, what should the base case be?

Write the recursive definition first, if it helps.

29.12.4 Balanced trees

Exercise 16.5.1 hint

1. If each subtree is balanced, do they necessarily have the same or similar heights?
2. If the tree is balanced, what do we know about the balance factors of its nodes?

Exercise 16.5.2 hint

The assumption is the same: one subtree is empty and the other has $n-1$ nodes. Use the complexity of the height function for each subtree. Intuitively, what's the complexity you expect?

Exercise 16.5.3 hint

If a tree is unbalanced, are some checks redundant?

29.12.5 Heapsort

Exercise 16.6.1 hint

Think of a concrete example, e.g. searching for 2 in the heaps shown.

Exercise 16.6.2 hint

What's the height of a complete tree, in terms of the input size n ?

Exercise 16.6.3 hint

Is there any input for which no swaps are done up or down the tree?

29.13 Graphs 1

29.13.1 Modelling with graphs

Exercise 17.1.1 hint

There are more than five edges.

Exercise 17.1.3 hint

What's the binary relation between items in a sequence?

Exercise 17.1.5 hint

One graph may not be enough.

29.13.2 Basic concepts

Exercise 17.2.1 hint

Which properties of nodes or edges can you use?

29.13.3 Edge list representation

Exercise 17.3.1 hint

Look closely at the examples.

Exercise 17.3.2 hint

Remember that removing a node also removes its edges.

29.13.4 Adjacency matrix representation

Exercise 17.4.2 hint

For most real-world graphs, what is larger: n or e ?

29.13.5 Adjacency list representation

Exercise 17.5.2 hint

The operations no longer do linear searches over adjacency lists, because Python sets support constant-time membership checks.

29.13.6 Classes for graphs

Exercise 17.6.3 hint

If all nodes have the same degree, the number of different degree values is one.

29.13.7 Traversing a graph

Exercise 17.7.1 hint

In a connected undirected graph, any node can be reached from any other.

Exercise 17.7.2 hint

Look at the path graph and the generated graph.

29.14 Greed

29.14.1 Minimum spanning tree

Exercise 18.3.1 hint

You must keep track of the most recently added node.

Exercise 18.3.2 hint

Use the graph that *illustrates the TSP*.

Exercise 18.3.3 hint

Keep the priority queue as short as possible.

Exercise 18.3.4 hint

If all weights are the same, the graph can be seen as being unweighted.

29.14.2 Shortest paths

Exercise 18.4.1 hint

What is a shortest path in this case?

Exercise 18.4.2 hint

Once you have the tree, how can you extract the path to the end node? Can the construction of the tree stop early?

Exercise 18.4.3 hint

Think of your country. Which stations would you consider central and why, in terms of shortest paths?

29.15 Practice 2

29.15.1 Jousting

Exercise 19.1.1 hint

Which ADT can efficiently retrieve the two currently strongest knights?

Exercise 19.1.2 hint

When will the tournament take longest to conclude? What is the most efficient implementation of a priority queue?

Exercise 19.1.3 hint

What are the edge cases for the tests?

Use a *Python heap* for the priority queue. Represent each item and its priority as a tuple (strength, index). Remember that Python provides min-heaps. You must negate the strength when using it as the priority.

29.15.2 Dot product

Exercise 19.2.1 hint

How many candidates (rearranged sequences) does the search generate and test? What's the complexity of testing whether a candidate has the lowest dot product?

Exercise 19.2.3 hint

Sort the input sequences from best to worst choice. [Section 4.6](#) introduced Python's sorting function.

29.15.3 Beams

Exercise 19.3.1 hint

As usual, think of the smallest inputs and output but also of other cases.

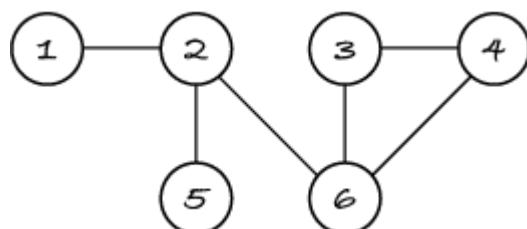
Exercise 19.3.2 hint

There's really only one sensible choice ...

Exercise 19.3.3 hint

Consider the following graph. Node 2 is weak: it isn't part of a triangle. There's no edge between any of its neighbours 1, 5 and 6. Node 6 is strong because there's an edge between two of its neighbours, forming a triangle with nodes 3 and 4.

Figure 19.3.1



What algorithmic strategy can you use to find all weak nodes?

Exercise 19.3.4 hint

For each node, the exhaustive search goes through pairs of neighbours. When does the search do the least or most work? Remember that the input is a set of edges.

29.15.4 Up and down

Exercise 19.4.1 hint

Linear search is efficient, but there's a more efficient way.

Exercise 19.4.2 hint

Read again the *binary search variants* section. How can you find out if the middle element is in the ascending or in the descending part?

Exercise 19.4.3 hint

You can write the code recursively or iteratively.

29.15.5 A knight goes places

Exercise 19.5.1 hint

What are the smallest inputs?

Exercise 19.5.2 hint

Like a spreadsheet, this grid ‘hides’ a graph ...

Exercise 19.5.3 hint

Which general graph problem is most similar to this problem? How must you adapt the corresponding graph algorithm for this problem?

Exercise 19.5.4 hint

To create the graph, define an auxiliary inner function that checks if the knight can move between two given squares. If so, add the corresponding edge to the graph.

To compute the shortest path, copy the *BFS code* and modify it.

29.16 Graphs 2

29.16.1 Undirected graph components

Exercise 21.1.1 hint

Do you need to compute all of the graph’s components? Do you need to know which node is in which component, i.e. do you need a map?

Exercise 21.1.2 hint

Think what can happen if a node and its edges are removed. How could you ‘measure’ how much disruption removing a node causes?

29.16.2 Directed graph components**Exercise 21.2.1 hint**

Simply follow the definition of weakly connected components: make an undirected copy of the input graph and return its connected components.

29.16.3 Topological sort**Exercise 21.3.1 hint**

Look up the complexity of `in_degree` in [Section 17.9](#).

Exercise 21.3.2 hint

Think what happens with a concrete problem instance, like A → B → C → A.

Exercise 21.3.3 hint

1. Check if the output of `topological_sort` doesn’t have all the graph’s nodes. Make the check as efficiently as possible.

Exercise 21.3.4 hint

1. Remember that nodes in a strongly connected component are mutually reachable.
2. What’s the worst-case scenario for deciding whether a digraph is cyclic?

29.16.4 State graphs**Exercise 21.4.1 hint**

1. How many nodes per square are generated?
2. How many neighbours does each state node have at most?

29.16.5 Practice**Exercise 21.5.1 hint**

In terms of graph concepts, what’s an island? Alternatively, go through the graph problems you know and for each one think if this problem could be rephrased as that graph problem.

Exercise 21.5.2 hint

There are three kinds of connected components. Which one is best suited to this problem?

Exercise 21.5.3 hint

Do you need a node for each square?

Exercise 21.5.4 hint

What does the *algorithm for connected components* return? From that output, how can you obtain the number of islands?

29.17 Backtracking

29.17.1 Back to the TSP

Exercise 22.5.1 hint

Use method `graph.weight(node1, node2)` to obtain the weight of an edge.

Exercise 22.5.3 hint

The item is the new node being considered, the candidate is a path and the local constraint is that there's an edge from the last node of the path to the new node.

29.17.2 Back to the knapsack

Exercise 22.7.1 hint

What are the conditions for a candidate to be the new best solution?

29.18 Dynamic Programming

29.18.1 Longest common subsequence

Exercise 23.2.2 hint

The cache is a dictionary where the keys are problem instances (here, pairs of strings), so you will need to write `cache[(left, right)]` to set or access a value in the dictionary.

Exercise 23.2.4 hint

The LCS of `left[1:]` and `right[r:]` must be stored in `cache[l][r]`.

29.18.2 Knapsack

Exercise 23.3.1 hint

Start at index 0 and increase it every recursive call.

Exercise 23.3.2 hint

1. The DAG only has two kinds of edges. If each node is a list–capacity pair, then the edges are:

- $(items, c) \leftarrow (tail(items), c)$ to indicate the first item was skipped
- $(items, c) \leftarrow (tail(items), c - weight(head(items)))$ to indicate the first item was taken (added to the knapsack).

If you write each node as an index–capacity pair, then the edges are:

- $(i, c) \leftarrow (i + 1, c)$
- $(i, c) \leftarrow (i + 1, c - weight(i))$.

A node where the list is empty, or the index is the number of items, has no in-neighbours as there are no further items to consider.

2. A subproblem is solved as often as there are paths from it to the root problem.

Exercise 23.3.3 hint

Like for the LCS problem, create a matrix with `None` in all cells. The auxiliary function is like the original recursive function, but stores the result in `cache[index][capacity]` and then returns it.

Exercise 23.3.4 hint

The body of the nested loops is like for the top-down approach but recursive calls to `knapsack(..., ...)` are replaced with `cache[...][...]`.

29.19 Practice 3

29.19.1 Safe places

Exercise 24.1.5 hint

Look up the complexity of Dijkstra's algorithm in [Section 18.5.1](#).

Exercise 24.1.6 hint

Dijkstra's algorithm finds the shortest paths from *one* node to *all* other nodes. Think of a way of applying the algorithm only once.

29.19.2 Extra staff

Exercise 24.2.2 hint

In this problem, we must execute as many tasks as possible per step (week). Look again at [Kahn's algorithm](#).

It keeps a set of nodes with in-degree 0 and in each iteration picks *one* of the nodes, appends it to the topological sort and decreases the degree of its out-neighbours. If any out-neighbour degree decreases to zero, then it's added to the set.

For this problem, how many nodes must you pick in each iteration? Do you need to keep an extra collection of nodes?

Exercise 24.2.3 hint

Copy the [topological sort code](#) and modify it as needed, following the answer to the previous question.

29.19.3 Borrow a book

Exercise 24.3.1 hint

Think of a similar past problem.

Exercise 24.3.2 hint

What characterises a state the book can be in? Draw the state transition graph for the given weekly schedule. Once you have the graph, how can an algorithm find the fewest days from the book being with Alice on Monday to being with Celia?

Exercise 24.3.3 hint

Copy and modify the [code for Dijkstra's algorithm](#) so that it stops when a node (*borrower, day*) is reached and returns the distance of that node from the (*lender, Mon*) start node, instead of returning the tree of all shortest paths.

Write code that creates the [example graph](#). Test your modified algorithm with that graph.

Finally, if you have time, write an auxiliary function that takes a tabular schedule and returns the corresponding transition graph.



Info: These exercises are based on problem *Book Hand Delivery* from the 2011 Portuguese Inter-University Programming Marathon.

29.19.4 Levenshtein distance

Exercise 24.4.1 hint

Look at the recursive definition of a similar problem: the *longest common subsequence*.

Exercise 24.4.2 hint

The `min` function can be applied to *more than two values*.

Exercise 24.4.3 hint

For the base case of `left[1:]` being empty, how many characters must be inserted to turn it into `right[r:]`?

Exercise 24.4.4 hint

You may first draw a DAG of problem instance dependencies for some test input, but give a general explanation of why there are overlapping subproblems.

Exercise 24.4.5 hint

The cache is a matrix like for the *LCS problem* but contains integers (the edit distances for the subproblems) instead of strings.

Exercise 24.4.6 hint

Look at the recursive definition or at the code of the recursive or top-down solution: which problem instances may instance `edit(l, r)` depend on? In terms of matrix cells, what cells may `cache[l][r]` depend on and where are those cells in relation to row `l`, column `r`?

Exercise 24.4.8 hint

It's the matrix size multiplied by the worst-case complexity of filling each cell.

29.19.5 Higher and higher

Exercise 24.5.1 hint

What kind of problem is this? Can you recall a similar problem?

Exercise 24.5.3 hint

Copy code from the *best sequence* backtracking template and from my *Trackword* solution and modify it as necessary.

Exercise 24.5.4 hint

Change your perspective from generating candidates to solving subproblems and check if there are overlapping subproblems.

Exercise 24.5.5 hint

The length of the longest path from the square in row r , column c is recursively defined as follows.

- if (r, c) has no higher neighbours: $\text{length}(r, c) = 1$
- otherwise: $\text{length}(r, c) = 1 + \text{highest of}$
 - $\text{length}(ra, ca)$ for every adjacent square (ra, ca) with a higher number

First implement a purely recursive inner function `length(row, column)`. Function `higher` must call it for every square in the grid and return the largest of all lengths.

Then add a matrix cache to obtain a top-down dynamic programming solution, with `cache[r][c]` being the length of the longest path starting at square (r, c) .

The longest path from square S may go next to the square above, below, left or right of S, so there's no way of filling the cache with two nested loops that iterate over the rows and columns in a strict top-down left-right (or some other) fashion. You're therefore not expected to produce a bottom-up version.



Info: This is LeetCode problem 329. Dynamic programming solutions can be found in the Discuss tab next to the locked Solution tab.

29.20 Complexity classes

29.20.1 Tractable and intractable problems

Exercise 26.1.1 hint

What is the worst-case complexity of the most efficient algorithm for the interval scheduling problem?

Exercise 26.1.2 hint

Look at the complexity of the dynamic programming solution and at the size of the output in terms of the input size.

Exercise 26.1.3 hint

If the graph has $n > 4$ nodes and e edges, how many possible paths are there in the worst case?

Exercise 26.1.4 hint

Does the question make sense?

29.20.2 The P and NP classes

Exercise 26.2.1 hint

What kind of information would convince you that the answer is ‘yes’ for a given graph and integer?

29.20.3 Reductions

Exercise 26.3.1 hint

What is the value of n ?

Exercise 26.3.2 hint

Do you need to modify the input sequence? How is the sorted sequence transformed into the largest value?

29.21 Computability

29.21.1 Turing machine

Exercise 27.1.1 hint

This is a similar problem to the parity bit: the machine reads the input on the tape and writes an extra symbol at the end. Once the symbol is written, should you move the head?

Exercise 27.1.2 hint

Write the dictionary systematically: add one line for each state–symbol pair and think what should be the new state for that pair. Most entries in the map move the head right and write the symbol that was read.

Exercise 27.1.3 hint

Think small first: if the password could have the lowercase letter ‘b’, in addition to ‘a’ and ‘0’, what entries would you add and how many are they?

Exercise 27.1.4 hint

For this problem, the states represent what conditions have been satisfied, e.g. if the machine is in state ‘letter’, it has read at least one letter, but no digit yet.

29.21.2 The Church–Turing thesis

Exercise 27.2.1 hint

Can you still use ‘X’ to mark the last letter counted? What transition must you change to not restore the original input letters?

Exercise 27.2.2 hint

A symbol can be any Python value, it doesn’t have to be a single character.

29.21.3 Undecidability

Exercise 27.4.1 hint

Remember that reductions are *transitive*.

Exercise 27.4.2 hint

Note in which direction the reduction is applied.

CHAPTER 30

ANSWERS

This chapter contains the answers for all exercises throughout the book.

30.1 Numbers and sequence

30.1.1 Arithmetic operations

Exercise 2.2.1 answer

I would use the modulo operation. If $x \bmod 2 = 0$, then x is even, otherwise it's odd.

30.1.2 Expressions

Exercise 2.3.1 answer

It's wrong. According to the rules, $-2^3 = -(2^3) = -(2 \times 2 \times 2) = -8$.

The result happens to be the same because the exponent is odd. If it were even, then the results would differ in their sign, e.g. $-2^2 = -(2^2) = -4$, whereas $(-2)^2 = -2 \times -2 = 4$.

30.1.3 Assignments

Exercise 2.4.1 answer

Here's my solution.

1. let *price* be 100
2. let *tax rate* be 0.2
3. let *total price* be *price* \times (1 + *tax rate*)
4. print *total price*

You may have chosen a different price and different names, but make sure the names are descriptive. You may also have written

2. let *tax rate* be $20 / 100$
3. let *total price* be $\text{price} + \text{price} \times \text{tax rate}$

You may have remembered that code cells can display a value if it's the last line of the cell, and have omitted the word 'print' from step 4. I'm sorry to say that's wrong. An English description of an algorithm is an abstract description that can be translated to different computational environments, Jupyter notebooks being only one of them.

Exercise 2.4.2 answer

```
[1]: price = 100
tax_rate = 0.2
total_price = price * (1 + tax_rate)
print(total_price)
```

120.0

30.1.4 Functions in mathematics

Exercise 2.5.1 answer

Here's my solution; other variable names and currencies are possible. You could also represent the rate as an integer between 0 and 100.

Function: total price

Inputs: *price*, a real number; *vat rate*, a real number

Preconditions: $\text{price} > 0$; *price* is in euros; $0 \leq \text{vat rate} < 1$

Output: *total*, a real number

Postconditions: $\text{total} = \text{price} \times (1 + \text{vat rate})$; *total* is in euros

30.1.5 Functions in Python

Exercise 2.6.1 answer

```
[1]: def brick_volume(length: int, width: int, height: int) -> int:
    """Return the volume of a brick, given its dimensions.

    Preconditions:
    - length > 0; width > 0; height > 0
    - length, width and height are in millimetres
    Postconditions:
    - the output is length * width * height
    - the output is in cubic millimetres
    """
    return length * width * height
```

Exercise 2.6.2 answer

```
[1]: def total_price(price: float, vat_rate: float) -> float:
    """Return the total price, including VAT at the given rate.

    Preconditions:
    - price > 0 and price is in euros
    - 0 <= vat_rate < 1
    Postconditions:
    - the output is price * (1 + vat_rate)
    - the output is in euros
    """
    return price * (1 + vat_rate)

total_price(100, 0.2)
```

```
[1]: 120.0
```

30.1.6 Complexity

Exercise 2.7.1 answer

The algorithm does a fixed number of operations, namely four: two assignments and two multiplications. All these operations have constant complexity, hence so does the algorithm. The complexity is $\Theta(1)$. (You may also phrase it as the algorithm runs in constant time or the algorithm takes $\Theta(1)$ time.)

30.1.7 Run-times

Exercise 2.8.1 answer

$1^y = 1$ for all y , so the exponentiation might be optimised for this base and take constant time. This wouldn't allow us to see any growth in run-times.

It may well be that exponentiation is *not* optimised for a base of 1, and thus still shows the sub-linear growth. There's only way to find out: change the base to 1 in the code cell, run it and see what happens.

Exercise 2.8.2 answer

I chose the modulo operation.

```
[1]: left = 2222
right = 7777
%timeit -r 6 -n 2000000 left % right    # 4 digits
left = 22222222
right = 77777777
%timeit -r 6 -n 2000000 left % right    # 8 digits
```

(continues on next page)

(continued from previous page)

The run-times are similar, which suggests constant complexity. In fact, for all the above cases, we have $0 < \text{left} < \text{right}$ and thus $\text{left} \bmod \text{right} = \text{left}$, which means that the modulo can indeed be computed in constant time. Let's try $\text{left} > \text{right} > 0$.

(continues on next page)

(continued from previous page)

```
25.2 ns ± 0.282 ns per loop (mean ± std. dev. of 6 runs, 2,000,000 loops each)
53.8 ns ± 0.351 ns per loop (mean ± std. dev. of 6 runs, 2,000,000 loops each)
59.4 ns ± 0.277 ns per loop (mean ± std. dev. of 6 runs, 2,000,000 loops each)
71.7 ns ± 0.516 ns per loop (mean ± std. dev. of 6 runs, 2,000,000 loops each)
```

While the run-times do increase, they do not double as the number of digits doubles, which suggests that the modulo can be computed in less than linear time in the size of the operands. Anyhow, as for addition, we will treat modulo as a constant-time operation if the operands are 64-bit integers.

30.2 Booleans and selection

30.2.1 Booleans

Exercise 3.1.1 answer

Function: disjunction

Inputs: *left*, a Boolean; *right*, a Boolean

Preconditions: true

Output: *result*, a Boolean

Postconditions: *result* = false if and only if *left* = *right* = false

Exercise 3.1.2 answer

With only two Boolean values available, the output of the negation operation is uniquely determined by stating that it's different from the input.

Function: negation

Inputs: *value*, a Boolean

Preconditions: true

Output: *negated*, a Boolean

Postconditions: *negated* ≠ *value*

You may have used other input and output variable names. Note that

Postconditions: *negated* = not *value*

doesn't really define the negation operation: it just shows the notation. It's like saying 'the addition of two numbers is the sum of two numbers': it states that the sum is the result of adding numbers, but it doesn't define addition, what the operation does. If you don't know beforehand what addition is, then the statement doesn't help you.

Exercise 3.1.3 answer

It's not a good name, because 'phone state?' isn't a yes/no question. The variable name doesn't tell us which state is represented by which Boolean.

Exercise 3.1.4 answer

There are many two-way settings on a mobile phone. Some examples: the phone is in sound or vibration mode; the screen is in light or dark mode; Bluetooth is on or off.

Exercise 3.1.5 answer

```
[1]: True and not (True or True)  
[1]: False
```

Exercise 3.1.6 answer

The lowercase word `true` is interpreted as a variable name. Unless you defined (i.e. assigned a value to) a variable with that name, the interpreter will raise a name error.

As you can imagine, using variables named `true`, `FALSE`, etc. can lead to confusion and make your code harder to understand, so avoid such names.



Note: Avoid variable names that are similar to Python keywords.

Exercise 3.1.7 answer

`(true or ((not true) and false)) or (not false) =`

`(true or (false and false)) or true =`

`(true or false) or true =`

`true or true = true`

30.2.2 Decision problems

Exercise 3.2.1 answer

The table has four instances with flight mode off, so the three missing instances have flight mode on. This means the output is always false.

Case	<i>in flight mode</i>	<i>wifi on</i>	<i>data on</i>	<i>internet on</i>
flight mode and Wi-Fi	true	true	false	false
flight mode and data	true	false	true	false
only flight mode	true	false	false	false

Exercise 3.2.2 answer

```
[1]: def internet_connection(in_flight_mode: bool, wifi_on: bool, data_on: bool) -> bool:
    """Return whether there's a connection to the internet.

    Postconditions: the output is true if and only if
    (not in_flight_mode) and (wifi_on or data_on)
    """
    internet_on = (not in_flight_mode) and (wifi_on or data_on)
    return internet_on
```

```
[2]: internet_connection(True, True, False)
[2]: False
```

```
[3]: internet_connection(True, False, True)
[3]: False
```

```
[4]: internet_connection(True, False, False)
[4]: False
```

Exercise 3.2.3 answer

```
[1]: def internet_connection(in_flight_mode: bool, wifi_on: bool, data_on: bool) -> bool:
    internet_on = (not in_flight_mode) and (wifi_on or data_on)
    return internet_on

%timeit -r 3 -n 1000 internet_connection(False, True, False)
%timeit -r 3 -n 1000 internet_connection(False, True, True)

99.1 ns ± 1.18 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
 ↪each)
99.7 ns ± 1.82 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops
 ↪each)
```

30.2.3 Boolean expressions

Exercise 3.3.1 answer

left ≠ right

Exercise 3.3.2 answer

1. The errors are: a space between `>` and `=`; `but` is not a logical operation; `not` doesn't take an integer operand.
2. The syntactically correct expression is `-5 >= value > -20` and `value != 0`.
3. It can be simplified to `-5 >= value > -20` because if the value is in that interval, it can't be zero.

30.2.4 Classification problems

Exercise 3.4.1 answer

1. if $mark \geq 80$:
 1. let `pass` be 1
2. otherwise if $mark \geq 60$:
 1. let `pass` be 2
3. otherwise if $mark \geq 50$:
 1. let `pass` be 3
4. otherwise if $mark \geq 40$:
 1. let `pass` be 4
5. otherwise:
 1. let `pass` be 5

Exercise 3.4.2 answer

They are the same conditions as the original algorithm, just in a different order. They are still comprehensive and mutually exclusive: for each mark, exactly one condition is true. The algorithm is correct: it satisfies the postconditions.

Exercise 3.4.3 answer

The algorithm only produces two grades: merit and distinction. Step 1.1 classifies any mark below 80 as a merit and step 4.1 classifies the remaining marks as distinctions. Any mark below 60 is a counter-example because it shouldn't get a merit.

To correct the algorithm, the first condition can't be simplified. The upper bound of the fourth condition can be omitted due to the preconditions. The final condition can be omitted because it covers whatever cases remain.

1. if $60 \leq mark < 80$:
 1. let `pass` be 2
2. otherwise if $mark < 40$:
 1. let `pass` be 5

3. otherwise if $50 \leq \text{mark} < 60$:

 1. let *pass* be 3

4. otherwise if $80 \leq \text{mark}$:

 1. let *pass* be 1

5. otherwise:

 1. let *pass* be 4

As both examples show, checking for the category boundaries in a haphazard order doesn't allow for many simplifications.

30.2.5 Practice

Exercise 3.5.1 answer

1. It's a decision problem because it asks a yes/no question: can a phone make and receive calls?
2. The preconditions state the possible values of the signal strength.

Function: calls possible

Inputs: *flight mode*, a Boolean; *signal strength*, an integer

Preconditions: $0 \leq \text{signal strength} \leq 4$

Output: *make calls*, a Boolean

Postconditions: *make calls* = true if and only if *flight mode* = false and *signal strength* ≥ 2

You may have chosen different function and variable names. The problem statement doesn't preclude the signal strength to be internally represented by a real number, with only five values shown in the user interface. If typing \geq is not easy with your keyboard, you may write ' > 1 ' instead in this example.

Exercise 3.5.2 answer

The boundary values are one (signal isn't strong enough) and two (it's strong enough). I add a test case where the flight mode overrides the signal strength.

Case	<i>flight mode</i>	<i>signal strength</i>	<i>make calls</i>
weak signal	false	1	false
strong enough	false	2	true
flight mode on	true	4	false

Exercise 3.5.3 answer

1. let *make calls* be (not *flight mode*) and (*signal* ≥ 2)

Since the algorithm has a single step, it's not necessary to number it. A 'stop' instruction at the end is unnecessary too: an algorithm stops if it has no more instructions to execute.

Exercise 3.5.4 answer

```
[1]: def calls_possible(flight_mode: bool, signal_strength: int) -> bool:  
    """Return whether the phone can make and receive calls.  
  
    Preconditions: 0 <= signal_strength <= 4  
    Postconditions: the output is true if and only if  
        flight_mode is false and signal_strength >= 2  
    """  
    return (not flight_mode) and (signal_strength >= 2)
```

You may have omitted the redundant parentheses in the return statement.

```
[2]: calls_possible(False, 1) == False
```

```
[2]: True
```

```
[3]: calls_possible(False, 2) == True
```

```
[3]: True
```

```
[4]: calls_possible(True, 4) == False
```

```
[4]: True
```

Exercise 3.5.5 answer

1. My function definition is as follows:

Function: maximum

Inputs: *left*, a real number; *right*, a real number

Preconditions: true

Output: *largest*, a real number

Postconditions: *largest* = *left* if *left* > *right*, otherwise *largest* = *right*

Although the output can only be one of two values (either *left* or *right*), we can't treat them as if they were Booleans and use 'if and only if' in the postconditions. The following is wrong because it doesn't specify which of the infinitely many real numbers the output is when $left \leq right$.

Postconditions: *largest* = *left* if and only if *left* > *right*

2. My test table uses a mix of integers and decimal numbers and has one test case for each possible relationship between the inputs.

Case	<i>left</i>	<i>right</i>	<i>largest</i>
<i>left > right</i>	5	3.5	5
<i>left < right</i>	-5.3	3.5	3.5
<i>left = right</i>	-3	-3	-3

Exercise 3.5.6 answer

1. if *left > right*:
 1. let *largest* be *left*
2. else:
 1. let *largest* be *right*

Exercise 3.5.7 answer

```
[1]: def maximum(left: float, right: float) -> float:
    """Return the largest of the two values.

    Postconditions:
    return left if left > right, otherwise return right
    """
    if left > right:
        largest = left
    else:
        largest = right
    return largest
```

You may have written:

```
if left > right:
    return left
else:
    return right
```

My tests are:

```
[2]: maximum(5, 3.5) == 5
```

```
[2]: True
```

```
[3]: maximum(-5.3, 3.5) == 3.5
```

```
[3]: True
```

```
[4]: maximum(-3, -3) == -3
```

[4] : True

Exercise 3.5.8 answer

This is a decision problem, so the output is a Boolean. The calendar was introduced in October 1582, so the first year for which the leap year rule applies is 1583.

Function: is leap year

Inputs: *year*, an integer

Preconditions: *year* > 1582

Output: *leap*, a Boolean

Postconditions:

- if *year* mod 4 ≠ 0, then *leap* = false
- if *year* mod 4 = 0 and *year* mod 100 ≠ 0, then *leap* = true
- if *year* mod 100 = 0 and *year* mod 400 ≠ 0, then *leap* = false
- if *year* mod 400 = 0, then *leap* = true

If you have written the conditions differently, check that for every integer exactly one of the conditions is true.

The first full year of the Gregorian calendar is an edge case and hence included in the test table.

Case	input	output
divisible by 400	1600	true
divisible by 100 but not by 400	1700	false
divisible by 4 but not by 100	2020	true
smallest value (not divisible by 4)	1583	false

Exercise 3.5.9 answer

My first algorithm checks the mutually exclusive and comprehensive conditions separately.

1. if *year* mod 4 ≠ 0:
 1. let *leap* be false
2. if *year* mod 4 = 0 and *year* mod 100 ≠ 0:
 1. let *leap* be true
3. if *year* mod 100 = 0 and *year* mod 400 ≠ 0:
 1. let *leap* be false
4. if *year* mod 400 = 0:
 1. let *leap* be true

Next I use ‘otherwise’ to not re-check conditions that I know are false.

1. if $year \bmod 4 \neq 0$:
 1. let $leap$ be false
2. otherwise if $year \bmod 100 \neq 0$:
 1. let $leap$ be true
3. otherwise if $year \bmod 400 \neq 0$:
 1. let $leap$ be false
4. otherwise:
 1. let $leap$ be true

I notice that steps 3 and 4 assign the output variable to the opposite value of the condition in step 3, so I can simplify those steps by using negation.

1. if $year \bmod 4 \neq 0$:
 1. let $leap$ be false
2. otherwise if $year \bmod 100 \neq 0$:
 1. let $leap$ be true
3. otherwise:
 1. let $leap$ be not $year \bmod 400 \neq 0$

Step 3.1 can be simplified because ‘not different from’ is equivalent to ‘is equal to’.

1. if $year \bmod 4 \neq 0$:
 1. let $leap$ be false
2. otherwise if $year \bmod 100 \neq 0$:
 1. let $leap$ be true
3. otherwise:
 1. let $leap$ be $year \bmod 400 = 0$

You may have checked conditions in the opposite order and obtained:

1. if $year \bmod 400 = 0$:
 1. let $leap$ be true
2. otherwise if $year \bmod 100 = 0$:
 1. let $leap$ be false
3. otherwise:
 1. let $leap$ be $year \bmod 4 = 0$

Other variations of the above are possible.

Looking at the algorithm I see that the output is true when the year is divisible by 400 or when it's divisible by 4 but not 100. This leads to a much simpler algorithm:

1. let $leap$ be $year \bmod 400 = 0$ or ($year \bmod 4 = 0$ and $year \bmod 100 \neq 0$)

I can reformulate the postconditions accordingly, which I'll do in the Python docstring.

Exercise 3.5.10 answer

```
[1]: def is_leap_year(year: int) -> bool:  
    """Return whether the given year is a leap year.  
  
    Preconditions: year > 1582  
    Postconditions: return true if and only if  
    year is divisible by 4 but not by 100, or is divisible by 400  
    """  
    return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)
```

```
[2]: is_leap_year(1600) == True
```

```
[2]: True
```

```
[3]: is_leap_year(1700) == False
```

```
[3]: True
```

```
[4]: is_leap_year(2020) == True
```

```
[4]: True
```

```
[5]: is_leap_year(1583) == False
```

```
[5]: True
```

If you don't feel confident about if-elif-else statements, I suggest you try to code one or two of the intermediate algorithms.

30.3 Sequences and iteration

30.3.1 The Sequence ADT

Exercise 4.1.1 answer

If $values$ is empty, the precondition becomes $0 \leq index < 0$. There's no integer that can satisfy both inequalities. When the sequence is empty, no value for $index$ can satisfy the preconditions. The operation is therefore not defined for the empty sequence.

30.3.2 Strings

Exercise 4.2.1 answer

```
[1]: character = "6"
character in "0123456789"
[1]: True
```

Exercise 4.2.2 answer

1. This is a type error: indices must be integers.
2. This is an index error: there's no first character in an empty string.
3. This is correct: a character is a string of length 1.
4. This is a type error: the left operand must be a string.

Exercise 4.2.3 answer

1. In a best-case scenario, no characters need to be copied: the output is the empty string. This may happen because the input string s is empty or the number of times t isn't positive. In a worst-case scenario, each character is copied t times to the output string.
2. The best-case complexity is $\Theta(1)$. The worst-case complexity is $\Theta(|s| \times t)$.

Exercise 4.2.4 answer

For both strings of length 0 and 1, `middle` is zero, because it's half the length rounded down.

If the length is zero, the last line evaluates to `text[0:0] + text[0:0]`, which results in the empty string, because the start and end indices are the same.

If the length is one, the last line evaluates to `text[0:1] + text[0:0]`. The right operand is the empty string and the left operand is `text`.

In all cases the slices are well defined and don't raise an error.

Exercise 4.2.5 answer

Converting a number n to a string s takes linear time in the number of digits to be processed: $\Theta(|n|)$. In M269 we only use numbers that fit in 64 bits, so the number of digits is limited. We can assume the conversion takes constant time: $\Theta(1)$.

Exercise 4.2.6 answer

First solution:

```
[1]: text = "hello"
times = 3
(text + "...") * (times - 1) + text
[1]: 'hello...hello...hello'
```

This works when `times` is one, because repeated concatenation produces the empty string when the integer operand is zero.

Second solution:

```
[2]: text = "hello"
times = 3
((text + "...") * times)[:-3] # whole string without last 3 dots
[2]: 'hello...hello...hello'
```

30.3.3 Iteration

Exercise 4.3.1 answer

The change is to swap the start and end indices and add ‘down’.

1. let `text` be ‘hello’
2. for each `index` from $|text| - 1$ down to 0:
 1. print `text[index]`

The translation involves replacing ‘down’ by -1 and decrementing the end index.

```
[1]: text = "hello"
for index in range(len(text) - 1, -1, -1):
    print(text[index])
```

```
o
l
l
e
h
```

30.3.4 Linear search

Exercise 4.4.1 answer

This is one of many possible test tables.

Case	<i>password</i>	<i>is valid</i>
smallest input	“”	false
smallest valid input	‘a5’	true
no lowercase letters	‘MK 01908’	false
no digits	‘päßword’	false
no lowercase or digits	‘^+/-&?’	false
both and others	‘my_P455W0RD’	true

Exercise 4.4.2 answer

The algorithm resets both Booleans for each character and hence forgets whether it had already found a letter or digit. Step 4 only checks the validity of the last character in the string. No character is both a letter and a digit, so the output is always false, even for valid passwords.

Exercise 4.4.3 answer

Alice's algorithm stops when the password is valid, but doesn't assign a value to the output variable, so the postconditions are undefined and thus not satisfied.

Bob's has the same issue for only one input: the empty string. The loop (step 3) is skipped and no value is assigned to the output variable, because Bob moved Alice's step 4 to inside the loop. If Bob had an extra final step (outside the loop) that sets *is valid* to false to handle the empty string, he would have a correct algorithm.

Exercise 4.4.4 answer

The best-case scenario is passwords that have a lowercase letter and a digit as their first two characters. The loop executes two iterations, i.e. a fixed number of constant-time operations.

The algorithm does the most work if it goes through the whole input string. Two scenarios can lead to this. If the password is invalid, the conditions are never both true, and the loop can't stop early. Another worst-case scenario is for the password to be valid but the second condition only becomes true with the last character, e.g. '221b' or 'Milton Keynes MK7'.

Exercise 4.4.5 answer

Here's the more efficient version. I simplified the code slightly. I don't write the postconditions explicitly in the docstring: they're already captured by the description.

```
[1]: def valid_password(password: str) -> bool:
    """Return whether password has a digit and a lowercase letter."""
    has_letter = False
    has_digit = False
    for character in password:
        if "0" <= character <= "9":
            has_digit = True
        if "a" <= character <= "z":
            has_letter = True
        if has_letter and has_digit:
            return True
    return False
```

I also simplify the tests: `b == True` is equivalent to `b`, and `b == False` is equivalent to `not b` for Boolean `b`.

```
[2]: not valid_password("")
```

```
[2]: True
```

```
[3]: valid_password("a5")
```

```
[3]: True
```

```
[4]: not valid_password("MK 01908")
```

```
[4]: True
```

```
[5]: not valid_password("päßword")
```

```
[5]: True
```

```
[6]: not valid_password("^+-/&?")
```

```
[6]: True
```

```
[7]: valid_password("my_P455W0RD")
```

```
[7]: True
```

30.3.5 Tuples and tables

Exercise 4.5.1 answer

The table was ‘rotated’: the same constants can be used to index rows instead of columns. The row index is written first.

```
[1]: games_by_column = ( # each item is a column
    ("Board game", "Power Grid", "Vintage", "Pandemic"),
    ("Rating", 10, 8, 9),
    ("Owned", True, True, False),
)
```

```
GAME = 0
```

```
RATING = 1
```

```
OWNED = 2
```

```
games_by_column[RATING][3]
```

```
[1]: 9
```

Pandemic is the last game in the table so `games_by_column[RATING][-1]` is an alternative expression.

Exercise 4.5.2 answer

```
[1]: tic_tac_toe = (
    ('X', 'O', 'X'),
    (' ', 'X', ' '),
)
```

(continues on next page)

(continued from previous page)

```

        ('O', ' ', 'O')
    )

empty = 0
for row in tic_tac_toe:
    for space in row:
        if space == ' ':
            empty = empty + 1
print(empty)
3

```

Exercise 4.5.3 answer

The obvious representation of a 10×10 grid is a tuple with 10 nested tuples, each of length 10. This requires representing each pawn as a tuple with the row and column indices of its current position. Calculating the new row and column after rolling, say, 5 isn't trivial.

To simplify the calculation of where a pawn moves to, the board can be represented 'flat' as a tuple of length 100. The position of each pawn is given by an integer variable with a value from 0 to 99. Moving a pawn after a roll of n requires adding n to its current position.

To handle snakes and ladders, the board is represented as a tuple of integers, each indicating how much to move forwards (positive integer) or backwards (negative integer) when landing on that position. Most positions have a value of zero.

If there's a ladder from position $start$ to position $end > start$, then the positive integer at index $start - 1$ (because indices are zero-based) is $end - start$. If there's a snake from position $tail$ to position $head > tail$, then the negative integer at index $head - 1$ is $tail - head$.

The moral of this exercise is that the chosen data type doesn't have to represent faithfully how something looks like in the user interface. A two-dimensional grid doesn't have to be represented as a table.

30.3.6 Lists

Exercise 4.6.1 answer

Operation: reverse

Inputs/Outputs: $values$, a sequence

Preconditions: true

Postconditions: post-values = (pre-values[|pre-values| - 1], pre-values[|pre-values| - 2], ..., pre-values[1], pre-values[0])

Exercise 4.6.2 answer

Concatenating two sequences has linear complexity in the total size of the sequences, whereas appending takes constant time. Unless we don't want to modify the original sequence, appending is always more efficient than concatenating a single item.

Exercise 4.6.3 answer

Adding a row is a doddle. It takes a single operation: appending or inserting a list with the full row in the table. It's therefore best to insert the column first. Adding a column requires appending or inserting a value in each row.

```
[1]: games_by_row = [
    ['Board game', 'Rating', 'Owned'],
    ['Power Grid', 10, True],
    ['Vintage', 8, True],
    ['Pandemic', 9, False]
]

games_by_row[0].append('Price in £')      # add column to header row
games_by_row[1].append(37)                 # and every other row
games_by_row[2].append(47)
games_by_row[3].append(26)
games_by_row.insert(1, ['Ticket to Ride', 8, False, 28])  # add row
games_by_row      # display the new table

[1]: [['Board game', 'Rating', 'Owned', 'Price in £'],
      ['Ticket to Ride', 8, False, 28],
      ['Power Grid', 10, True, 37],
      ['Vintage', 8, True, 47],
      ['Pandemic', 9, False, 26]]
```



Info: Prices were taken from BoardGamePrices.co.uk on 10 September 2019.

30.3.7 Reversal

Exercise 4.7.1 answer

The original algorithm went through the original list from left to right, inserting each item at the start of the reverse list. This algorithm goes through the original list in the reverse order and appends each item to the reverse list.

1. let *reverse* be the empty sequence
2. for each *index* from $|values| - 1$ down to 0:
 1. append *values*[*index*] to *reverse*

Exercise 4.7.2 answer

Steps 1 and 2.1 take constant time and can be ignored. The loop is executed for each index of the input list, so the complexity is $\Theta(|values|)$. The new algorithm is linear in the length of the input list and thus more efficient than the original quadratic algorithm.

Exercise 4.7.3 answer

```
[1]: from algoesup import test

reverse_list_tests = [
    # case,           values,          reverse
    ('empty list',   [],              []),
    ('length 1',     [4],             [4]),
    ('length 2',     [5, True],       [True, 5]),
    ('length 5',     [5, 6, 7, 8, 9], [9, 8, 7, 6, 5]),
    ('same items',   [0, 0, 0],       [0, 0, 0])
]

def reverse_list_2(values: list) -> list:
    """Return the same items as values, in inverse order.

    This is a more efficient version of reverse_list.
    Postconditions: the output is
    [values[-1], values [-2], ..., values[1], values[0]]
    """
    result = []
    for index in range(len(values)-1, -1, -1):
        result.append(values[index])
    return result

test(reverse_list_2, reverse_list_tests)

Testing reverse_list_2...
Tests finished: 5 passed (100%), 0 failed.
```

Exercise 4.7.4 answer

Strings can't be modified, so appending is replaced by concatenation.

1. let *reverse* be the empty string
2. for each *index* from $|values| - 1$ down to 0:
 1. let *reverse* be *reverse* + *values*[*index*]

If you want additional practice, then create a new test table, copy the `reverse_list_2` function, rename it to `reverse_string`, change the header, docstring and body as necessary, and test it.

Exercise 4.7.5 answer

Concatenation copies the items in both operands, so step 2.1 has complexity $\Theta(|reverse| + 1) = \Theta(|reverse|)$. The loop is executed for each item in *values*, so the complexity is quadratic: $|values| \times \Theta(|reverse|) = \Theta(|values| \times |reverse|) = \Theta(|values| \times |values|) = \Theta(|values|^2)$.

Exercise 4.7.6 answer

The algorithm stops when both fingers meet (the sequence has odd length) or cross each other (the sequence has even length).

1. let *left index* be 0
2. let *right index* be $|values| - 1$
3. while *left index* < *right index*:
 1. *left item* = *values*[*left index*]
 2. *right item* = *values*[*right index*]
 3. let *values*[*left index*] be *right item*
 4. let *values*[*right index*] be *left item*
 5. let *left index* be *left index* + 1
 6. let *right index* be *right index* - 1

The first four steps in the loop can be simplified to:

1. *left item* = *values*[*left index*]
2. let *values*[*left index*] be *values*[*right index*]
3. let *values*[*right index*] be *left item*

but not to:

1. let *values*[*left index*] be *values*[*right index*]
2. let *values*[*right index*] be *values*[*left index*]



Info: TM112 Block 1 Activity 2.16 explains why.

Exercise 4.7.7 answer

```
[1]: def reverse_in_place(values: list) -> None:  
    """Reverse the order of the values.  
  
    Postconditions: post-values = [pre-values[len(pre-values) - 1],  
        ..., pre-values[1], pre-values[0]]  
    """
```

(continues on next page)

(continued from previous page)

```

left_index = 0
right_index = len(values) - 1
while left_index < right_index:
    left_item = values[left_index]
    values[left_index] = values[right_index]
    values[right_index] = left_item
    left_index = left_index + 1
    right_index = right_index - 1

reverse_list_tests = [
    # case,           values,          reverse
    ('empty list',   [],             []),
    ('length 1',     [4],            [4]),
    ('length 2',     [5, True],      [True, 5]),
    ('length 5',     [5, 6, 7, 8, 9], [9, 8, 7, 6, 5]),
    ('same items',   [0, 0, 0],      [0, 0, 0])
]

# `algoesup.test` can only test functions that return values,
# so I've written the testing code for you
for test in reverse_list_tests:
    name = test[0]
    values = test[1]
    reverse = test[2]
    reverse_in_place(values)
    if values != reverse:
        print(name, 'FAILED:', values, 'instead of', reverse)
print('Tests finished.')

```

Tests finished.

30.4 Implementing sequences

30.4.1 Developing data types

Exercise 6.3.1 answer

```
[1]: class Sequence:
    """The sequence ADT."""

    def has(self, item: object) -> bool:
        """Return True if and only if item is a member of self."""
        for index in range(self.length()):
            if self.get_item(index) == item:
                return True
```

(continues on next page)

(continued from previous page)

```
return False
```

I've added this method to the `m269_sequence.py` file, so that it can be used in the remainder of this chapter.

30.4.2 Bounded sequences

Exercise 6.4.1 answer

The messages show that the inserted numbers aren't copied to the right. Each new number is put at index 0, overwriting the number that was there. At the end, only the last inserted number (always 0) is in the sequence.

The problem is therefore in the loop that copies the items to the right, which is skipped when an item is appended and that's why no error occurs in that case.

The first line of the `insert` method iterates over `range(..., index, -1)`. The `range` constructor always excludes the value of the second argument, so the loop doesn't go down until `index`. It stops at `index + 1`. A loop 'from `x` down to `y`' must be translated to `range(x, y-1, -1)` in order to include `y`. (See the [summary](#) on iteration.) This is a typical 'off by one' error.



Note: Even if the algorithm in English is correct, its Python translation may not be. Always test your code.

Exercise 6.4.2 answer

Replace `index` with `index - 1` in the second `range` argument:

```
def insert(self, index: int, item: object) -> None:
    for position in range(self.size - 1, index - 1, -1):
        ...
```

30.4.3 Linked lists

Exercise 6.7.1 answer

If the item to be removed is the first one, i.e. `index = 0`, make the head point to the next node. Otherwise, iterate over the linked list to get a reference to the node before the one to be removed, i.e. at `index - 1`. Make that node point to the node after the one to be removed. This bypasses the node at position `index`, thus removing it from the chain.

Exercise 6.7.3 answer

Besides *head* pointing at the first node, we keep one pointer to the last node and update it when an item is appended.

30.5 Stacks and queues

30.5.1 Stacks

Exercise 7.1.1 answer

The item is appended to the items already in the stack, so no index is necessary, which simplifies the definition.

Operation: push

Inputs/Outputs: *values*, a stack

Inputs: *value*, an object

Preconditions: true

Postconditions: post-*values* = (pre-*values*[0], ..., pre-*values*[|pre-*values*| - 1], *value*)

Exercise 7.1.2 answer

Operation: pop

Inputs/Outputs: *values*, a stack

Inputs: none

Preconditions: $0 < |\text{values}|$

Postconditions: post-*values* = (pre-*values*[0], ..., pre-*values*[|pre-*values*| - 2])

Function: peek

Inputs: *values*, a stack

Preconditions: $0 < |\text{values}|$

Output: *value*, an object

Postconditions: *value* is the *n*-th item of *values*, with $n = |\text{values}|$

Exercise 7.1.3 answer

The run-time may be better with linked lists because they don't require resizing. On the other hand, dynamic arrays waste less memory.

30.5.2 Using stacks

Exercise 7.2.1 answer

All stack operations and comparisons take constant time.

The least work is done when the loop stops in its first iteration. The best-case scenario is a closing bracket at the start of the string. The best-case complexity is constant.

The algorithm does the most work if a decision can only be made after reaching the end of the string, in step 3. A worst-case scenario is a balanced string. The worst-case complexity is linear in the length of the string.

Other worst-case scenarios, with the same complexity obviously, are strings where the final character is a mismatched closing bracket and strings that have more opening than closing brackets, which otherwise match.

Exercise 7.2.2 answer

```
[1]: %run -i ./m269_stack

from algoesup import test

def is_balanced(text: str) -> bool:
    """Check if all brackets in text are balanced.

    Preconditions: each bracket in 'text' is one of (, ), [,
    """
    opened = Stack() # opened but not closed brackets
    for character in text:
        if character in "(":
            opened.push(character)
        elif character == ")":
            if opened.size() > 0 and opened.peek() == "(":
                opened.pop()
            else:
                return False
        elif character == "]":
            if opened.size() > 0 and opened.peek() == "[":
                opened.pop()
            else:
                return False
    return opened.size() == 0

is_balanced_tests = [
    # case,           text,                                balanced
    ['no text',      '',                                 True],
    ['no brackets',  'brackets are like Russian dolls', True],
    ['matched',      '(3 + 4)',                          True],
    ['mismatched',  '(3 + 4]',                           False],
    ['not opened',   '3 + 4]',                           False],
    ['not closed',   '(3 + 4',                           False],
    ['wrong order',  'close ) before open (',          False],
    ['nested',       '((([])))',                         True],
    ['nested pair',  'items[(i - 1):(i + 1)]',          True]
]
```

(continues on next page)

(continued from previous page)

```
]  
  
test(is_balanced, is_balanced_tests)  
Testing is_balanced...  
Tests finished: 9 passed (100%), 0 failed.
```

Exercise 7.2.3 answer

If there are no square brackets, it's pointless to keep a stack that contains just '(': we only need to know how many of them are in the stack. An integer counter can thus replace the stack. The algorithm becomes:

Initialise an integer counter with value zero. It will count how many parentheses were opened but not yet closed. Iterate over the string. Increment the counter for each '(' and decrement it for each ')'. If the counter becomes negative, then a ')' has no matching '(', so stop immediately: the parentheses are unbalanced. If the end of the string is reached, then the parentheses are balanced if and only if the counter is zero.



Note: Only use a collection data type if you need to track the individual items.

Exercise 7.2.4 answer

Each operator applies to the two immediately preceding operands. In other words, the *last* values before the operator are the *first* ones to be processed. The stack will thus contain only operands, not operators.

Exercise 7.2.5 answer

Create an empty stack that will contain the operands. Iterate over the sequence representing the expression. If the item is a number, push it on the stack. If the item is a character, peek and pop the right operand (because it was pushed last), then peek and pop the left operand. Apply the operation given by the operator character to both operands and push the result back on the stack. After the sequence is processed, there's only one number in the stack: the value of the expression.

Exercise 7.2.6 answer

```
[1]: %run -i ../m269_stack  
  
from algoesup import test
```

(continues on next page)

(continued from previous page)

```
def evaluate(expression: list) -> int:
    """Return the value of the given postfix expression.

    Preconditions:
    - each item in the input list is a natural number, '-' or '*'
    - the input represents a valid non-empty postfix expression
    """
    values = Stack()
    for item in expression:
        if item == "-":
            right = values.pop()
            left = values.pop()
            values.push(left - right)
        elif item == "*":
            right = values.pop()
            left = values.pop()
            values.push(left * right)
        else: # it's an operand
            values.push(item)
    return values.peek()

evaluate_tests = [
    # case,           expression,      value
    ["3 * 4",        [3, 4, "*"],     12],
    ["3 - 4",        [3, 4, "-"],     -1],
    ["3 - 4 * 5",   [3, 4, "*", "-"], -17],
    ["(3 - 4) * 5", [3, 4, "-", 5, "*"], -5],
    ["(3 - 4) * (5 - 6)", [3, 4, "-", 5, 6, "-", "*"], 1],
    ["no operation", [4],             4]
]

test(evaluate, evaluate_tests)

Testing evaluate...
Tests finished: 6 passed (100%), 0 failed.
```

30.5.3 Queues

Exercise 7.3.1 answer

Tasks are not necessarily completed in the order they were added. Therefore, I would use a sequence, as it is the most flexible ADT.

Exercise 7.3.3 answer

If the front item is in the last node, we must remove the node when dequeuing the item. For that, we need to get a reference to the penultimate node and then replace its `next` pointer with the null pointer to indicate that the linked lists now ends there. Finding the penultimate node requires traversing the linked list, which takes linear time.

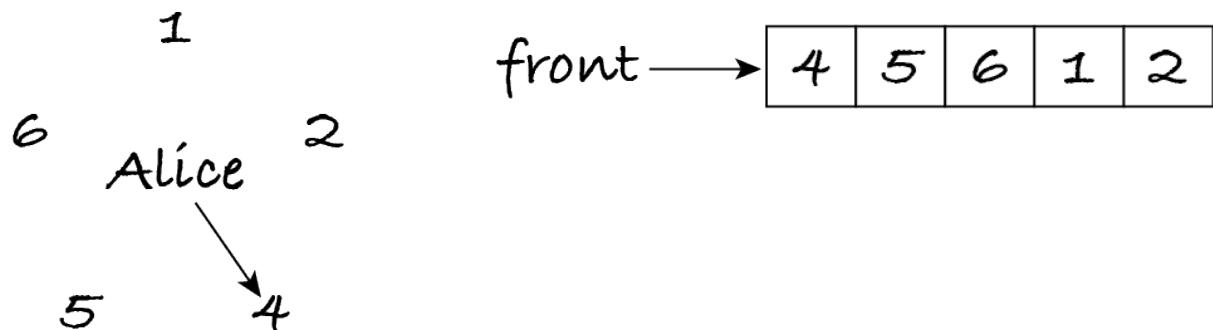
This is yet another example of how the way we structure data affects the operations' complexity.

Exercise 7.3.4 answer

At any point in time, the queue contains the children (more specifically, the number of each child) that haven't been counted out yet. The child at the front is the one Alice is currently pointing to. The rest of the queue contains the remaining children, in clockwise order.

Here's an example of 6 children in a circle and the corresponding queue. For illustration, child 3 has already been counted out and Alice is pointing at child 4.

Figure 7.3.1



Exercise 7.3.5 answer

Create a queue with numbers 1, 2, ..., n in that order, i.e. with 1 at the front: the number of the child Alice is pointing to first. For each of $n - 1$ rounds, repeatedly remove the number at the front and add it to the back of the queue. This simulates Alice pointing to the next child. Repeat this 27 times for each round. On the 28th time, don't add the number back to the queue: this simulates the child leaving the circle. After all rounds, the front item is the number of the last remaining child.



Info: This scheme of processing the item at the front of a queue and putting it back at the end is used to implement [round-robin scheduling](#).

Exercise 7.3.6 answer

Here's one possible translation to Python of the algorithm's outline in the previous exercise.

```
[1]: %run -i ../m269_queue

from algoesup import test

def counting_rhyme(n: int) -> int:
    """Return which child from 1 to n remains last in the circle.

    Preconditions: n > 0
    """
    circle = Queue()
    for child in range(1, n + 1):
        circle.enqueue(child)
    for round in range(n - 1):
        # in each round, the child on syllable 28 leaves the circle
        for syllable in range(27):
            circle.enqueue(circle.dequeue())
        circle.dequeue()
    return circle.front()

counting_rhyme_tests = [
    # case,           n,   last child
    ['1 child',     1,      1],
    ['2 children',  2,      1],
    ['3 children',  3,      2]
]

test(counting_rhyme, counting_rhyme_tests)

Testing counting_rhyme...
Tests finished: 3 passed (100%), 0 failed.
```

Exercise 7.3.7 answer

The first loop iterates n times to create the circle. The outer loop iterates $n - 1$ times to simulate the $n - 1$ rounds and the inner loop iterates 27 times. All methods take constant time, and so does the inner loop.

The algorithm boils down to two linear-time loops, one after the other, and so the overall complexity is linear in n . In Big-Theta notation: $\Theta(n) + \Theta(n - 1) \times 27 \times \Theta(1) = \Theta(n) + \Theta(n) = \Theta(2n) = \Theta(n)$ because constants are ignored.

30.5.4 Priority queues

Exercise 7.4.1 answer

Inserting is appending (constant time) followed by sorting (linear in the best case and quadratic in the worst), so the latter determines the overall complexity.

Operation	Complexity
insert	best $\Theta(pq)$, worst $\Theta(pq ^2)$
find max	$\Theta(1)$
remove max	$\Theta(1)$

Exercise 7.4.2 answer

The priorities are in ascending order, so we do a linear search for the first priority that is higher than the priority of the new item.

If existing items have the same priority as the new item, then the new item gets inserted in the first index of the existing items, so that it is removed last from the priority queue.

In other words, as we go through the array of existing priorities, we skip all those that are lower than the new item's priority. If we reach the end of the array, the index is equal to the length: the new item will be appended. Here's one algorithm:

1. for each *index* from 0 to $|priorities| - 1$:
 1. if $priorities[index] \geq priority$:
 1. stop
 2. let *index* be $|priorities|$

Unfortunately, it's not a good one for this context. We usually translate 'stop' to a `return` statement, but here we can't return from the `insert` method at this point: we need to use the value of `index`.

Here's a better algorithm, with the added benefit of making its gist (skip lower priorities) clearer.

1. let *index* be 0
2. while $index < |priorities|$ and $priorities[index] < priority$:
 1. let *index* be *index* + 1

This algorithm uses Python's short-circuit conjunction to avoid indexing the array beyond its end. It's an instructive exercise to rewrite the algorithm to not rely on short-circuiting.

Exercise 7.4.3 answer

```
[1]: # this code is also in m269_priority.py
```

```
class ArrayPriorityQueue:
    def insert(self, item: object, priority: object) -> None:
        index = 0
        while index < len(self.priorities) and self.
            ↪priorities[index] < priority:
                index = index + 1
```

(continues on next page)

(continued from previous page)

```
self.items.insert(index, item)
self.priorities.insert(index, priority)
```

Exercise 7.4.4 answer

If the new item has to be inserted at index i , then it takes i iterations to find the position and another $|priorities| - i$ iterations to push the remaining priorities and items one position up. The complexity is thus always $\Theta(|priorities|)$.

Operation	Complexity
new	$\Theta(1)$
length	$\Theta(1)$
insert	$\Theta(pq)$
find max	$\Theta(1)$
remove max	$\Theta(1)$

Exercise 7.4.5 answer

Two dynamic arrays waste twice the memory when they're below capacity, and require twice the copying operations when resizing. Likewise, inserting an item and its priority takes twice the work after the index has been computed. So using two lists instead of a list of pairs is a really bad idea.

Exercise 7.4.6 answer

The insert operation appends the new priority–item pair.

The find max operation does a linear search over the whole array for the item with the highest priority, keeping track of the first it finds, as that is the oldest with that priority.

The remove max operation finds the max-priority item and removes it, copying the remaining priority–item pairs one position down.

Exercise 7.4.7 answer

A FIFO queue would force us to add events and their reminders in chronological order to the calendar.

Exercise 7.4.8 answer

For a min-priority queue we need to put the items in descending order. The `insert` method must skip all items with a higher (rather than a lower) priority. The only change is to replace the second `<` with `>`.

The question only asks about changes to the code, but you'd also need to change any docstrings and comments accordingly.

30.6 Unordered collections

30.6.1 Maps

Exercise 8.1.1 answer

The keys are the priorities, and the values are normal queues. In this way, all items of the same priority are in a single queue, with the oldest item at the front.

Exercise 8.1.2 answer

Find max:

Iterate over the keys, doing a linear search for the highest key value (highest priority). Access the front item of the queue associated with that key.

Remove max:

Find the highest priority as in find max. Remove the front item of the associated queue. If it becomes empty, remove the highest priority and the queue from the dictionary.

Add:

See if the given priority is a key in the dictionary. If it is, add the given item to the back of the associated queue. If it isn't, add a new key-value pair, where the key is the priority and the value is a new FIFO queue with just the given item.

Exercise 8.1.3 answer

A map with only two items is not very useful and requires more memory than using just two separate variables for the values.

Exercise 8.1.4 answer

1. If the priorities are known in advance, we can use a lookup table, possibly with a hash function (if priorities aren't integers). If the hash function takes constant time, so will the three operations.
2. Even without knowing the priorities in advance, in most real-life cases the number of priorities is a small number, so operations that are linear in the number of priorities take effectively constant time. Unless the priority queue never grows much, using a map is preferable.

This exercise shows that the most obvious data representation, e.g. a sorted sequence for a priority queue, is not always the best.

30.6.2 Dictionaries

Exercise 8.2.1 answer

In this way, the dictionary would store only one translation for each word.

Exercise 8.2.2 answer

Create an empty dictionary. For each Portuguese word–translations pair in the original dictionary, and for each English word in the translations list, check if the English word is a key in the new dictionary. If it is, add the Portuguese word to the list of translations; if it isn't, add a new entry where the key is the English word and the value is a list only containing the Portuguese word.

Exercise 8.2.3 answer

```
[1]: from algoesup import test

def invert(original: dict) -> dict:
    inverted = dict()
    for (word, translations) in original.items():
        for translation in translations:
            if translation not in inverted:
                inverted[translation] = [word]
            else:
                inverted[translation].append(word)
    return inverted
```

30.6.3 Hash tables

Exercise 8.3.1 answer

If the slots only contained values it would be impossible to know which of them is associated to the key being looked up. For example, if the second slot only contained 407 and 312, how would the lookup operation know which extension is Bob's and which is Alice's?

Exercise 8.3.2 answer

By storing the hash values, they don't have to be recomputed when the table is resized.

Exercise 8.3.3 answer

The load factor only influences the mean number of key–value pairs per slot. The core algorithm to find, add, replace and remove key–value pairs, namely hashing and modulo followed by a linear search, works no matter how long each slot is.



Info: This question is adapted from exercise R-10.6 of the book *Data Structures and Algorithms in Python* by Goodrich, et al.

30.6.4 Sets

Exercise 8.4.1 answer

The even primes are prime – odd and the odd primes are prime & odd.

Exercise 8.4.2 answer

The union is computed by first adding all items of a to the resulting set, which takes $\Theta(|a|)$, and then adding all members of b , which takes $\Theta(|b|)$.

The difference takes $\Theta(|a|)$ time because it does at most two constant-time operations per member of a : check if it's in b and add it to the output set if it isn't.

The intersection operation computes the common members, which by definition are in both sets. The operation goes through the smaller set, checking if each member is in the larger set. If it is, it's added to the result. The operation obtains and compares the size of both sets and then, assuming a is the smaller set, does $|a|$ membership operations in b and at most $|a|$ addition operations to the resulting set. All these operations take constant time, so the operation is linear in the size of the smaller of both sets.

Exercise 8.4.3 answer

1. The expression constructs a potentially large set, if the intersection has many members, only to check if it's non-empty. This wastes time and memory.
2. Compare the sizes of both sets to determine the smaller one. Iterate over it, checking if each member is in the larger set. If it is, the sets aren't disjoint: stop and return false. After iterating over the whole set, return true: no common member was found.
3. This approach doesn't construct a set, thus saving memory. A worst-case scenario is when the sets are disjoint: the algorithm is linear in the size of the shortest set, like the expression. In a best-case scenario, a common item is found after a few iterations: the best-case complexity is constant.

Exercise 8.4.4 answer

The algorithm keeps a set of the schools which already have a certificate. If a team's school isn't in that set, then the team gets a certificate and the school is added to the set.

```
[1]: def certificates(ranking: list) -> list:
    best_teams = []
    awarded_schools = set()
    for team in ranking:
        school = team[:-1] # school name without digit
        if school not in awarded_schools:
            awarded_schools.add(school)
            best_teams.append(team)
    return best_teams
```

(continues on next page)

(continued from previous page)

```
certificates_tests = [
    # case,           ranking,                  certificates
    ('3 schools',   ['C1', 'B2', 'B1', 'A1', 'C2'],  ['C1', 'B2', 'A1']),
    # new tests:
    ('1 team',      ['A1'],                      ['A1']),
    ('1 school',    ['C3', 'C1', 'C2'],            ['C3']),
    ('1 team per school', ['C1', 'B1', 'A1'],  ['C1', 'B1', 'A1'])
]
```



Info: This exercise is based on problem [ICPC Awards](#).

30.7 Practice 1

30.7.1 Pangram

Exercise 9.1.1 answer

The provided tests already cover the smallest output (empty string). There should be two tests for the largest output (all letters are missing): one for when the input is the empty string (smallest input) and another for when the input has no letter from ‘a’ to ‘z’ (but possibly other letters).

Exercise 9.1.2 answer

A straightforward algorithm checks whether each letter from ‘a’ to ‘z’ occurs in *text*. If not, it’s appended to *missing*, which starts empty.

1. let *missing* be the empty string
2. for each *letter* in ‘abc...xyz’:
 1. if *letter* isn’t in *text*:
 1. let *missing* be the concatenation of *missing* and *letter*

Step 2.1 does a linear search over the whole text for each letter. The order and frequency of the characters in *text* don’t matter, so the appropriate data type to represent the text is a set. This reduces *text* to its unique characters, i.e. a much smaller collection in the typical case, making step 2.1 more efficient.

1. let *missing* be the empty string
2. let *occurring* be the set of characters in *text*
3. for each *letter* in ‘abc...xyz’:
 1. if *letter* isn’t in *occurring*:
 1. let *missing* be the concatenation of *missing* and *letter*

This makes me think of a second algorithm that uses set operations to directly compute the missing letters. The resulting set must be converted to a sorted string.

1. let *occurring* be the set of characters in *text*
2. let *all* be the set of letters from ‘a’ to ‘z’
3. let *absent* be *all* – *occurring*
4. let *ordered* be *absent* as a sorted list
5. let *missing* be the empty string
6. for each *letter* in *ordered*:
 1. let *missing* be the concatenation of *missing* and *letter*

Step 4 can be implemented with Python’s `sorted` function.

Exercise 9.1.3 answer

The shortest output (*missing* is the empty string) occurs when *text* is a pangram. The longest output (*missing* is the string ‘abc..xyz’) occurs when *text* hasn’t any of ‘a’ to ‘z’. However, creating a string of at most 26 characters takes constant time, so the length of the output doesn’t affect the complexity.

My first algorithm always does 26 linear searches over the string *text*: in the best case it has constant complexity (when all letters occur in the first 26 positions of *text*) and in the worst case it has complexity $\Theta(|text|)$.

My second algorithm first goes through *text* to create a set of its characters: this has complexity $\Theta(|text|)$. Then it executes 26 membership operations on that set, which take constant time.

My third algorithm first creates two sets, one with the 26 lowercase letters, the other with *text*’s characters: this has complexity $\Theta(|text|)$. All the remaining operations (set difference, sorting, constructing *missing*) take constant time: they handle at most 26 letters.

To sum up, the first algorithm has constant complexity in the best case and linear complexity in the worst case. The other two algorithms always have linear complexity.

Exercise 9.1.4 answer

The first algorithm seems the best: it can take constant time only and doesn’t use additional memory. However, for each letter missing in *text*, the linear search goes through the whole, potentially very long, string.

The second algorithm goes through *text* only once, to create the set of characters. Although the set uses extra memory and, in its Python implementation, requires hashing of all characters in *text* and then hashing all 26 letters to look them up, this may be better than doing several long linear searches.

The third algorithm uses even more memory (a set of letters and the difference set) and does more operations (set difference and sorting), so I doubt it will be the most efficient.

To sum up, assuming that *text* can be very long and be missing several letters, I think the second algorithm is the most efficient.



Note: The best- and worst-case complexities don't account for typical cases and for the actual number of operations executed, which may be important factors when deciding between algorithms.

Exercise 9.1.5 answer

```
[1]: from algoesup import test

LETTERS = "abcdefghijklmnopqrstuvwxyz"

def missing_letters(text: str) -> str:
    """Return all English alphabet letters that aren't in text.

    Preconditions: all letters in text are lowercase
    Postconditions: the output are the letters from a to z that
    don't occur in text, in alphabetical order
    """
    occurring = set(text)
    missing = ""
    for letter in LETTERS:
        if letter not in occurring:
            missing = missing + letter
    return missing

PANGRAM = "the quick brown fox jumps over the lazy dog."

missing_letters_tests = [
    # case,           text,                  missing
    ['pangram',     PANGRAM,              ''],
    ['no vowels',   'bcd fgh jklmn pqrst vwxyz', 'aeiou'],
    # new tests:
    ['no text',      '',                  LETTERS],
    ['no a-z',       'Θ(1)',             LETTERS]
]

test(missing_letters, missing_letters_tests)

Testing missing_letters...
Tests finished: 4 passed (100%), 0 failed.
```

30.7.2 Election

Exercise 9.2.1 answer

There are at least three more edge cases: the smallest input (1 vote); 1 vote per candidate; a single candidate with all votes (more than one).

The first provided test (all candidates are tied) reminds me to add a typical case: some, but not all, candidates are tied. The smallest such case is when two of three candidates are tied.

Exercise 9.2.2 answer

The order in which votes were cast is irrelevant, only their frequency. The ADT of choice is a map, of candidates to the number of votes they got. A linear search over the key-value pairs will find the key (candidate) with the highest value (votes). Here's the outline of the algorithm:

Create an empty map of candidates (strings) to votes (integers). For each occurrence of a candidate in the input list, increment their vote count. After processing the input, do a linear search over all candidate-vote pairs, keeping the pair with the highest number of votes. If a candidate ties for the highest votes so far, set the best candidate to 'round 2'. At the end of the search, return the best candidate.

Exercise 9.2.3 answer

The first phase of the algorithm (creating the map) goes through all votes, whereas the linear search goes through the candidates to find the winner.

In the worst case, there are as many candidates as votes (each got 1 vote). The algorithm therefore goes twice through the votes. The worst-case complexity is $\Theta(|votes|)$.

Exercise 9.2.4 answer

This approach does some processing of the counted votes (sorting) to then obtain the winner or tie in constant time.

However, whereas my approach goes once through the candidates to find the winner or a tie, sorting can take in the worst case quadratic time in the number of candidates. In the best case, it takes linear time and hence is not better than my approach.

Exercise 9.2.5 answer

```
[1]: from algoesup import test

def winner(votes: list) -> str:
    results = dict() # map candidate (str) to their votes (int)
    for candidate in votes:
        if candidate in results:
            results[candidate] = results[candidate] + 1
        else:
            results[candidate] = 1
```

(continues on next page)

(continued from previous page)

```

winner_votes = 0
for (candidate_name, candidate_votes) in results.items():
    if candidate_votes > winner_votes:
        winner_votes = candidate_votes
        winner_name = candidate_name
    elif candidate_votes == winner_votes:
        winner_name = "round 2"
return winner_name

winner_tests = [
    # case,           votes,                  name
    ['2 of 2 tied', ['Alice', 'Bob', 'Bob', 'Alice'], 'round 2'],
    ['1 of 2 wins', ['Alice', 'Bob', 'Alice', 'Alice'], 'Alice'],
    ['1 of 3 wins', ['Bob', 'Chan', 'Chan', 'Alice'], 'Chan'],
    # new tests:
    ['1 vote',       ['Chan'],                'Chan'],
    ['1 vote per candidate', ['Bob', 'Chan', 'Alice'], 'round 2'],
    ['1 candidate',  ['Bob'] * 5,              'Bob'],
    ['2 of 3 tied', ['Al', 'Bob', 'Chan', 'Al', 'Bob'], 'round 2']
]

test(winner, winner_tests)

```

Testing winner...

Tests finished: 7 passed (100%), 0 failed.



Info: This exercise is based on problem Recount.

30.7.3 Voucher

Exercise 9.3.1 answer

The best way to find pairs that cost the voucher's amount is to organise products by price. If the current product costs n , I need to quickly know if any previously seen products cost $voucher - n$. I thus use a map of prices to products. No product is free, so products costing the voucher's amount or more can be ignored as they can't be matched with any other product.

Create an empty map of prices to product sequences. Create an empty set $results$. Iterate over $store$. For each product $current$, if its cost $price$ is less than $voucher$, append $product$ to the map's entry for $price$. Then add $(previous, current)$ to $results$ for every product $previous$ in the map's entry for $voucher - price$. The added pairs must not be $(current, previous)$ due to the postconditions.

Follow up question: does it matter whether we add the current product to the map before or after

finding the matching products? Hint: consider the last test case.

Exercise 9.3.2 answer

In a best-case scenario there are no matching products. The algorithm simply iterates over all products and processes each in constant time, assuming all set and map operations are $\Theta(1)$. Each product is either ignored (if it costs too much) or added to the map, but there are no products in the matching entry.

Exercise 9.3.3 answer

In a worst-case scenario, all products cost half the amount of the voucher and thus match each other, leading to $|store|^2$ product pairs. Since the output has a quadratic number of items, producing the output takes quadratic time in the size of the store.

Exercise 9.3.4 answer

The map's keys can only be 1, 2, ..., $voucher - 1$. So using a lookup table is more efficient than using a hash table. The table's first position (index 0) won't be used.

Exercise 9.3.5 answer

```
[1]: def pairs(store: list, voucher: int) -> set:
    results = set()
    products_by_price = []
    for price in range(voucher):
        products_by_price.append([])

    for product in store:
        price = product[1]
        if price < voucher:
            products_by_price[price].append(product)
        for matched in products_by_price[voucher - price]:
            results.add((matched, product))
    return results
```



Info: A simpler version of this problem, with a single matching pair, is a classic: it's [Two Sum](#) on LeetCode, *Store Credit* in Google Code Jam Africa 2010 (problem statement no longer available) and [Report Repair](#) in Advent of Code 2020.

30.7.4 Trains

Exercise 9.4.1 answer

It's a decision problem because the output is a Boolean.

Exercise 9.4.2 answer

At least a test for the smallest input (1 wagon) should be added. The last provided test suggests adding the opposite test: move the front wagon to the end.

Exercise 9.4.3 answer

Incoming and outgoing trains can be modelled as queues because wagons are attached to the back and detached from the front. The wagons in the station form a stack: the last one in is the first one out. (The layout of the track was a hint to use queues and a stack.) The algorithm is:

Create an empty stack to represent the station. Create an empty queue for the incoming (east) train: enqueue all integers from 1 to *wagons*. Convert the *outgoing* list to a queue, representing the outgoing (west) train.

While the outgoing queue isn't empty (i.e. there are still wagons to process) check if the wagon at the top of the station stack is the next expected outgoing wagon (front of the queue). If so, the wagon is removed from the station and the outgoing queue: it has been dealt with. If they don't match, the only alternative is to move the next incoming wagon in the east to the station. If the incoming queue is empty, we ran out of options and the outgoing permutation can't be produced.

Exercise 9.4.4 answer

The algorithm first creates two queues of length *wagons*, then it enters the main loop.

A worst-case scenario is when the main loop doesn't exit early. In that case the outgoing permutation is possible and the loop executes $2 \times \text{wagons}$ times: each wagon has to enter from the east and leave to the west, when it's removed from the outgoing queue.

Since all queue and stack operations take constant time, the complexity is $\Theta(4 \times \text{wagons}) = \Theta(\text{wagons})$ or $\Theta(|\text{outgoing}|)$.

Exercise 9.4.5 answer

I use two queues and a stack.

```
[1]: %run -i ./m269_queue
%run -i ./m269_stack
from algoesup import test

def rearrange(wagons: int, outgoing: list) -> bool:
    """Check if the incoming train can be rearranged into outgoing.

    Preconditions:
    - wagons > 0
    - outgoing is a permutation of the numbers from 1 to wagons
    """
    east = Queue()
    for wagon in range(1, wagons + 1):
        east.enqueue(wagon)
```

(continues on next page)

(continued from previous page)

```

west = Queue()
for wagon in outgoing:
    west.enqueue(wagon)

station = Stack()
while west.size() > 0:
    if station.size() > 0 and west.front() == station.peek():
        # move wagon out of station, tick it off the outgoing
        west.dequeue()
        station.pop()
    elif east.size() == 0:
        # no more incoming wagons available
        return False
    else:
        # move incoming wagon to station
        station.push(east.dequeue())
return True

rearrange_tests = [
    # case,           wagons,  outgoing,      rearrange?
    ['keep order',   3,       [1, 2, 3],      True],
    ['invert',       3,       [3, 2, 1],      True],
    ['swap',         3,       [1, 3, 2],      True],
    ['move to front', 5,       [5, 1, 2, 3, 4], False],
    # new tests:
    ['1 wagon',     1,       [1],            True],
    ['move to back', 4,       [2, 3, 4, 1],  True]
]

test(rearrange, rearrange_tests)

Testing rearrange...
Tests finished: 6 passed (100%), 0 failed.
    
```

Both queues can be represented in a much simpler way, each with a single integer: the front wagon of the incoming `east` queue and the index of the next wagon in `outgoing` to be sent off. While this is more efficient in terms of memory and run-time, it's an implementation optimisation. Conceptually, the incoming and outgoing trains are queues, and thinking in terms of such abstractions helps to get a grip on a problem.



Note: When solving a problem, try first to model it in terms of one or more ADTs.



Info: These exercises are based on problem Rails.

30.7.5 SMS

Exercise 9.5.1 answer

The completions operation is described as returning the first three words that match the prefix, when words are in descending score order. So all I need is a sequence of words in that order. Implementing a sequence as a dynamic array uses the least additional memory.

Initialisation: Create a sequence of all score–word pairs, sort it by descending score and then remove the scores, i.e. replace each score–word pair with the word, as they’re no longer needed. This reduces the memory used.

In the best case, the initialisation does three iterations over all words, so the best-case complexity is linear in the number of words. In the worst case, the sorting takes quadratic time in the number of words.

Completions: Create an empty suggestions sequence. Do a linear search over the sorted sequence of words. For each word, if it starts with the given prefix, append it to the suggestions. Stop on finding three suggestions or on reaching the end of the word sequence.

The linear search takes constant time in the best case (when the matching words are at the start of the sequence) and linear time in a worst case, e.g. if there are fewer than three matching words.

Exercise 9.5.2 answer

```
[1]: class SMS:
    def __init__(self, filename: str) -> None:
        data = []
        with open(filename, "r") as infile:
            for line in infile:
                pair = line.split()
                word = pair[0]
                score = int(pair[1])
                data.append((score, word))
        data.sort(reverse=True)
        for index in range(len(data)):
            data[index] = data[index][1] # discard the score
        self.words = data

    def completions(self, prefix: str) -> list:
        suggestions = []
        length = len(prefix)
        for word in self.words:
            if word[:length] == prefix:
                suggestions.append(word)
```

(continues on next page)

(continued from previous page)

```
if len(suggestions) == 3:
    return suggestions
return suggestions
```



Info: The files contain the 100 (respectively 10,000) most common words of all the books in Project Gutenberg, as of 16 April 2006. Project Gutenberg is a collection of thousands of free e-books, mostly in English and out of copyright. The files hence include archaic English words, like ‘thou’ and ‘thy’. I created the files from [Wiktionary data](#). I rounded down all fractional frequencies, converted all words to lowercase, and replaced `mother+`'s with `mother'`s. Wiktionary has more [word frequency lists](#), including for languages other than English.

Exercise 9.5.3 answer

For my approach, the worst-case complexity is linear when there are fewer than three matches, like for prefix ‘said’. As the vocabulary increases 100 times, from 100 to 10,000 words, I expect the same increase in the worst run-time. Indeed, this happened on my computer: about 9.5 µs for 100 words and about 950 µs for 10,000 words. To find completions among 100,000 words, I expect a further 10-fold increase in the worst-case run-time: 9500 µs. That’s about 10 ms, well below the 50 ms limit. (A mobile phone’s processor may take longer, of course.)



Note: If you’re pressed for time, try the simplest approach that works: it may be good enough.

Exercise 9.5.4 answer

The key idea (pun intended) to achieve constant time is to look up (rather than compute) the words for each prefix, using a map with the prefixes as keys and the completions (sequences of at most three strings) as values. If the prefix is in the map, then the operation returns the associated completions, otherwise it returns an empty sequence: no word has that prefix and so there are no completions.

The initialisation operation does all the work. Each word is associated to all its prefixes. The sequence of score–word pairs for each prefix is then sorted in descending score order, trimmed to the first three suggestions and the scores removed. In this way the map is ready for the completions operation to use.

1. let *suggestions* be an empty map
2. for each *word* and *score* read from the file:
 1. for each *length* from 0 to $|word|$:
 1. let *prefix* be $word[0:length]$

2. if *prefix* isn't in *suggestions*:
 1. let *suggestions(prefix)* be the empty sequence
 3. append (*score*, *word*) to *suggestions(prefix)*
3. for each *prefix* in *suggestions*:
 1. let *words* be *suggestions(prefix)* in descending order
 2. let *words* be *words[0:3]*
 3. for each *index* of *words*:
 1. let *words[index]* be *words[index][1]*
 4. let *suggestions(prefix)* be *words*

The loop in step 2.1 starts at *length* 0 to produce the empty prefix.

All steps take linear or constant time because they do at most a constant number l of operations where l is the length of a word. The slowest step is 3.1: sorting can take from constant time in the best case, if there's only one word for a given prefix, to $\Theta(n^2)$ in the worst case, if all n words have the same prefix. That case actually occurs: all words have the empty prefix. So the whole algorithm takes $\Theta(n)$ in the best case, when that entry is already sorted because the words are in descending score order in the file, and $\Theta(n^2)$ in the worst case (when we have no such luck).

Note that each word of length l has $l + 1$ prefixes (from the empty string to the word itself) and hence occurs $l + 1$ times in the map when step 3 starts. This uses quite a lot more memory than the first approach, but in exchange we can return the completions of any prefix in constant time.

The initialisation can be improved to use less memory and run always in linear time in the number of words, i.e. $\Theta(n)$. I'll leave that as an optional exercise. Hint: keep the sequences in the map bounded.

30.8 Exhaustive search

30.8.1 Linear search, again

Exercise 11.1.1 answer

To avoid copying the input collection, we must put each candidate item back after testing it. We can enqueue each item back into a queue, because it doesn't affect the next candidate to be dequeued, but putting an item back in a priority queue won't allow us to access the candidates with lower priority.

Exercise 11.1.2 answer

False. It has linear complexity only if generating and testing each candidate takes constant time.

Exercise 11.1.3 answer

The solution is the pattern in Section 5.2, which appends the items as they're found to an output sequence.

In step 1, replace 'empty collection' with 'empty sequence'. In step 2.1.1 replace 'add' with 'append'.

Exercise 11.1.4 answer

In step 1, replace 'empty collection' with 'empty set'.

Exercise 11.1.5 answer

The following doesn't change any of the existing steps; it only adds new steps to keep a collection of the solutions found so far.

1. let *best* be the first of *candidates*
2. let *solutions* be an empty collection
3. for each *candidate* in *candidates*:
 1. if *candidate* is better than *best*:
 1. let *best* be *candidate*
 2. let *solutions* be a collection just with *best*
 2. otherwise if *candidate* is as good as *best*:
 1. add *candidate* to *solutions*

Step 3.1.2 creates the solutions collection afresh, with the new best candidate.

Again, the output collection could be a sequence to keep solutions in the order they're found, or a set to avoid duplicate solutions.

Exercise 11.1.6 answer

The best case is when we can stop immediately after the first candidate. This happens when the first (and thus every) candidate costs more than £20. The worst case is the same as for the unsorted input: all candidates satisfy the conditions.

30.8.2 Factorisation

Exercise 11.2.1 answer

No. Computing the set of all factors just to check if there are two is a waste of time and memory.

Exercise 11.2.2 answer

If $n = 1$, the algorithm can immediately return false. For every prime $n > 1$, 1 and n are the two factors. The algorithm searches for a third factor from 2 to $\text{floor}(\sqrt{n})$ and returns false if it finds one. If the end of the loop is reached, n is prime.

30.8.3 Constraint satisfaction

Exercise 11.3.1 answer

Assuming 100 ns per iteration, for $product = 336$, the algorithm will take 30 seconds:

```
[1]: NS_PER_S = 1000**3 # nanoseconds in a second  
(2 * 336 + 1) ** 3 * 100 // NS_PER_S  
[1]: 30
```

For $product = 1320$, it will take 30 minutes:

```
[2]: NS_PER_M = 60 * NS_PER_S # nanoseconds in a minute  
(2 * 1320 + 1) ** 3 * 100 // NS_PER_M  
[2]: 30
```

30.8.4 Searching permutations

Exercise 11.4.1 answer

Under the given assumption, any tour has the same cost as the reversed tour, e.g. tours (0, 1, 2, 3, 0) and (0, 3, 2, 1, 0). Since the algorithm generates all permutations of the intermediate places, it will generate the reversed tours.

Exercise 11.4.2 answer

Not generating the symmetric reverse tours eliminates half of the candidates. However, reducing the run-time by a constant factor doesn't change the complexity.

30.8.5 Searching subsets

Exercise 11.5.1 answer

We generate feature subsets from the largest to the smallest and as soon as we find a feasible product (set of features), we output it.

1. for each k from $|features|$ down to 1:
 1. for each k -combination $product$ of $features$:
 1. if $\text{feasible}(product, incompatible)$:
 1. let $compatible$ be $product$
 2. stop

Exercise 11.5.2 answer

Generate each subset of items and test if their total weight doesn't exceed the rucksack capacity. If it doesn't, add the items' values. If the total value exceeds the highest value so far, update the solution to be the current candidate subset.

Note that every subset has to be generated, including the empty subset, for the edge case in which each item weighs more than the knapsack's capacity.

30.8.6 Practice

Exercise 11.6.1 answer

Converting to a set removes duplicates, so if the size decreases, the original list had duplicates.

Exercise 11.6.2 answer

We must search for a pair of identical items. If such a pair exists, the sequence has duplicates, otherwise it hasn't.

The candidates are therefore all pairs of items. If A isn't equal to B, then B isn't equal to A so we don't need to generate symmetric pairs. To generate pairs, we use two nested loops that iterate over all possible indices. To avoid generating symmetric pairs, the second index is larger than the first.

1. for each *first* from 0 to $|items| - 2$:
 1. for each *second* from *first* + 1 to $|items| - 1$:
 1. if $items[first] = items[second]$:
 1. let *duplicates* be true
 2. stop
 2. let *duplicates* be false

This is another example of a generate-and-test algorithm where the test involves more than one item.

Exercise 11.6.3 answer

All operations within the loops take constant time, so the complexity just depends on how many candidates are generated and tested.

In a best-case scenario, the first two items are duplicates and the algorithm stops after the first iteration of each loop. The best-case complexity is constant.

In a worst-case scenario, no items are duplicates and the algorithm generates and tests all non-symmetric pairs. If there are n items, there are roughly $n^2 / 2$ such pairs, so the worst-case complexity is quadratic.

Exercise 11.6.4 answer

The algorithm first sorts the input sequence: this brings all duplicate items together. Then it does a linear search, from the second item onwards, for an item that is equal to previous one. If such an item exists, a duplicate has been found and the algorithm stops. If the linear search ends without finding such an item, there are no duplicates.

Exercise 11.6.5 answer

It's an optimisation problem in which the total cost is being maximised, subject to a constraint.

Exercise 11.6.6 answer

This problem is a simplified version of the 0/1 knapsack, where there are only values instead of values and weights. The voucher amount plays the role of the knapsack's capacity.

The algorithm sets the best difference to infinity and the solution to the empty set. Then it generates all subsets and for each one computes the difference between the voucher amount and the subset's price. If the difference isn't negative and is lower than the best one, then the best difference and the solution are updated.

Exercise 11.6.7 answer

We can substantially reduce the search space by removing all products costing more than the voucher, before starting to generate the subsets. Each product removed halves the search space!

We can stop as soon as we get a subset of products that adds up exactly to the voucher amount, as no subset will be better.

We can also stop if all k -combinations exceed the voucher amount. It's pointless to continue searching: if any k products exceed the amount, so will any $k + 1$ products, any $k + 2$ products, etc. Checking if all k -combinations exceed the amount requires a Boolean flag, because we want to know if several candidates, together, satisfy some condition.

Exercise 11.6.8 answer

The candidates are all possible slices of t of the same length as s . For each index $start$ from 0 to $|t| - |s|$, the algorithm generates the slice $t[start:start + |s|]$ and tests if the slice is equal to s . If so, the algorithm stops with output true: s occurs in t at index $start$. If the loop reaches the end, s doesn't occur in t and the output is false.

Exercise 11.6.9 answer

In the worst case, s isn't a substring of t . My algorithm will compute $|t| - |s|$ slices, each taking $\Theta(|s|)$. The worst-case complexity is $\Theta((|t| - |s|) \times |s|)$.

Usually t is much longer than s , e.g. s is a word and t is a document, so the complexity can be simplified to $\Theta(|t| \times |s|)$.

Exercise 11.6.10 answer

It's a CSP: we want all 4-digit integers that satisfy certain constraints.

Exercise 11.6.11 answer

One approach uses four nested loops, each iterating from 1 to 9. The PIN is computed from the four digits (1000 times first digit plus 100 times second digit, etc.) and an auxiliary function tests if the PIN and the digits satisfy the conditions: all digits are different and divide the PIN. If they do, the PIN is added to the solutions set.

Another approach generates each 4-subset from the set $\{1, \dots, 9\}$ to guarantee by construction that all digits are different, and then generates all permutations of those four digits. Each such PIN is tested for divisibility by its digits.



Info: This exercise is adapted from problem Heir's Dilemma.

Exercise 11.6.12 answer

The problem has no input: everything about the problem is known in advance. Since there's no dependence on an input value, the complexity is constant for both algorithms. A different way to arrive at the same conclusion is to realise that both approaches do a fixed number of constant-time operations.

30.9 Recursion

30.9.1 Recursion on integers

Exercise 12.2.1 answer

We're told the integer is positive and we must decide whether it's even. So $\text{lowest} = 1$ and the output is a Boolean.

- if $n = 1$: $\text{even}(n) = \text{false}$
- if $n > 1$: $\text{even}(n) = \text{not even}(n-1)$

Exercise 12.2.2 answer

```
[1]: from algoesup import test

def even(n: int) -> bool:
    """Return True if and only if n is divisible by 2.

    Preconditions: n > 0
    """

```

(continues on next page)

(continued from previous page)

```
if n == 1:
    return False
else:
    return not even(n - 1)

even_tests = [
    # case,           n,      even?
    ('smallest even', 2,     True),
    ('larger even',   100,   True),
    # your tests here
    ('smallest input', 1,    False),
    ('larger odd',    45,    False)
]

test(even, even_tests)
```

Testing even...
Tests finished: 4 passed (100%), 0 failed.

30.9.2 Length of a sequence

Exercise 12.3.1 answer

- if *numbers* is empty: $\text{sum}(\text{numbers}) = 0$
- otherwise: $\text{sum}(\text{numbers}) = \text{sum}(\text{tail}(\text{numbers})) + \text{head}(\text{numbers})$.

30.9.3 Inspecting sequences

Exercise 12.4.1 answer

```
[1]: def maximum(numbers: list) -> int:
    """Return the largest number in numbers.

    Preconditions: numbers is a non-empty list of integers
    """
    if len(numbers) == 1:
        return head(numbers)
    else:
        return max(head(numbers), maximum(tail(numbers)))
```

Exercise 12.4.2 answer

The answers to the usual two questions are:

1. The smallest input is the empty sequence and the output is true.

2. Sequence S is ascending if the tail is ascending and if the head is smaller than the next number, which is the head of the tail.

I can only compare a number to the next one if there are at least two in the sequence, so I need to handle the cases of zero and one items separately, as base cases.

- if numbers is empty: $\text{ascending}(\text{numbers}) = \text{true}$
- if $\text{length}(\text{numbers}) = 1$: $\text{ascending}(\text{numbers}) = \text{true}$
- if $\text{length}(\text{numbers}) > 1$: $\text{ascending}(\text{numbers}) = \text{ascending}(\text{tail}(\text{numbers}))$ and $\text{head}(\text{numbers}) < \text{head}(\text{tail}(\text{numbers}))$.

The output is the same for both base cases, so they can be merged into one:

- if $\text{length}(\text{numbers}) \leq 1$: $\text{ascending}(\text{numbers}) = \text{true}$
- if $\text{length}(\text{numbers}) > 1$: $\text{ascending}(\text{numbers}) = \text{ascending}(\text{tail}(\text{numbers}))$ and $\text{head}(\text{numbers}) < \text{head}(\text{tail}(\text{numbers}))$.

A tail-recursive definition is also possible, by adding a second base case:

- if $\text{length}(\text{numbers}) \leq 1$: $\text{ascending}(\text{numbers}) = \text{true}$
- otherwise if $\text{head}(\text{numbers}) \geq \text{head}(\text{tail}(\text{numbers}))$: $\text{ascending}(\text{numbers}) = \text{false}$
- otherwise: $\text{ascending}(\text{numbers}) = \text{ascending}(\text{tail}(\text{numbers}))$.

Exercise 12.4.3 answer

```
[1]: from algoesup import test

%run -i ../m269_rec_list


def value(items: list, index: int) -> object:
    """Return the value at position index of items.

    Preconditions: 0 <= index < len(items)
    """
    if index == 0:
        return head(items)
    else:
        return value(tail(items), index - 1)

value_tests = [
    # case,           items,       index,   value
    ('shortest input', [5],      0,        5),
    ('access last',   [3, 4, 1], 2,        1),
    ('access middle', [3, 4, 1], 1,        4)
]
```

(continues on next page)

(continued from previous page)

```
test(value, value_tests)
Testing value...
Tests finished: 3 passed (100%), 0 failed.
```

30.9.4 Creating sequences

Exercise 12.5.1 answer

```
[1]: def positive(numbers: list) -> list:
    """Return a new list of all positive numbers in the input.

    Preconditions: all elements of numbers are integers or floats
    Postconditions:
        the output's elements are in the same order as in the input
    """
    if is_empty(numbers):
        return []
    else:
        solutions = positive(tail(numbers))
        if head(numbers) > 0:
            solutions = prepend(head(numbers), solutions)
        return solutions

positive_tests = [
    # case,           numbers,           positive
    ('shortest input', [],                []),
    # new tests:
    ('no positives',  [-1, 0, -3],       []),
    ('all positive',   [1, 3],            [1, 3]),
    ('mixed signs',   [3, 0, -1, 2],     [3, 2])
]
```

Exercise 12.5.2 answer

- if $items$ is empty: $\text{reverse}(items) = items$
- otherwise: $\text{reverse}(items) = \text{append}(\text{reverse}(\text{tail}(items)), \text{head}(items))$

Exercise 12.5.3 answer

The insight is similar to the indexing operation: inserting an item in the n -th position of a sequence is the same as inserting it in the $n-1$ -th position of the tail, and inserting in position 0 means to prepend it.

- if $index$ is 0: $\text{insert}(items, item, index) = \text{prepend}(item, items)$

- otherwise: $\text{insert}(\text{items}, \text{item}, \text{index}) = \text{prepend}(\text{head}(\text{items}), \text{insert}(\text{tail}(\text{items}), \text{item}, \text{index} - 1))$

30.9.5 Avoiding slicing

Exercise 12.6.1 answer

1. if $\text{start} = \text{end}$:
 1. let solutions be ()
2. otherwise:
 1. let solutions be $\text{search}(\text{candidates}, \text{start} + 1, \text{end})$
 2. if $\text{candidates}[\text{start}]$ satisfies conditions:
 1. let solutions be $\text{prepend}(\text{candidates}[\text{start}], \text{solutions})$

Exercise 12.6.2 answer

- if $\text{start} + 1 = \text{end}$: $\text{maximum}(\text{numbers}, \text{start}, \text{end}) = \text{numbers}[\text{start}]$
- otherwise: $\text{maximum}(\text{numbers}, \text{start}, \text{end}) = \max(\text{numbers}[\text{start}], \text{maximum}(\text{numbers}, \text{start} + 1, \text{end}))$

Exercise 12.6.3 answer

```
[1]: def maximum(numbers: list) -> int:
    """Return the largest number in numbers.

    Preconditions: numbers is a non-empty list of integers
    """
    def in_slice(start: int, end: int) -> int:
        """Return the largest integer in slice numbers[start:end].

        Preconditions: 0 <= start < end <= len(numbers)
        """
        if start + 1 == end:
            return numbers[start]
        else:
            return max(numbers[start], in_slice(start + 1, end))

    return in_slice(0, len(numbers))
```

30.9.6 Multiple recursion

Exercise 12.7.1 answer

I divide the sequence into halves. The results for each half are combined with disjunction. The algorithm requires two base cases, as when splitting a sequence into one third and two thirds. It's

not possible to split sequences of length 0 and 1 into two shorter sequences, so those are the base cases.

1. if $start = end$:
 1. let $exists$ be false
2. otherwise if $start + 1 = end$:
 1. let $exists$ be $items[start] = item$
3. otherwise:
 1. let $middle$ be $start + \text{floor}((end - start) / 2)$
 2. let $exists\ left$ be $\text{has}(items, item, start, middle)$
 3. let $exists\ right$ be $\text{has}(items, item, middle, end)$
 4. let $exists$ be $exists\ left$ or $exists\ right$

Another way of spotting that the algorithm needs two bases cases is to realise that with the first base case alone the algorithm always returns false. An algorithm for a decision problem must be able to return either Boolean value.

A more efficient version wouldn't recur into the right half if the item is in the left one.

Exercise 12.7.2 answer

```
[1]: def has(items: list, item: object) -> bool:
    """Return True if and only if item is a member of items."""

    def in_slice(start: int, end: int) -> bool:
        """Return True if and only if item is in slice items[start:end].

        Preconditions: 0 <= start <= end <= len(items)
        """
        if start == end:
            return False
        elif start + 1 == end:
            return items[start] == item
        else:
            middle = start + (end - start) // 2
            exists_left = in_slice(start, middle)
            exists_right = in_slice(middle, end)
            return exists_left or exists_right

    return in_slice(0, len(items))
```

30.10 Divide and conquer

30.10.1 Variable decrease

Exercise 13.3.1 answer

The GCD of two positive numbers a and b is a positive factor of both, so it's at least 1 and can't be larger than a or b . The range of candidates is thus 1 to $\min(a, b)$.

Since we want to find the greatest of them, we generate and test the candidates in reverse sorted order. The algorithm iterates over the integers from $\min(a, b)$ down to 1 and stops when a candidate divides both input numbers. Since 1 divides any integer, the algorithm is guaranteed to stop.

30.10.2 Binary search

Exercise 13.4.1 answer

If the sequence is in the opposite order, we search the opposite half, i.e. swap steps 3.4.1 and 3.5.1: search the right half if *item* is smaller than the middle member, otherwise search the left half.

Exercise 13.4.2 answer

All operations take constant time and the recursive call is made on one half, so

- $T(0) = \Theta(1)$
- $T(n) = T(n / 2) + \Theta(1)$.

As indicated for the *exponentiation algorithm*, this recursive definition leads to $T(n) = \Theta(\log n)$.

30.10.3 Binary search variants

Exercise 13.5.1 answer

```
[1]: def first_positive(numbers: list) -> int:
    def in_slice(start: int, end: int) -> int:
        """Return the first positive number within numbers[start:end].

        Preconditions: 0 <= start < end <= len(items)
        """
        if end - start == 1:
            return numbers[start]
        elif end - start == 2:
            if numbers[start] > 0:
                return numbers[start]
            else:
                return numbers[start + 1]
```

(continues on next page)

(continued from previous page)

```

else:
    middle = start + (end - start) // 2
    if numbers[middle] > 0:
        return in_slice(start, middle + 1) # include middle
    else:
        return in_slice(middle + 1, end)

return in_slice(0, len(numbers))

```

Exercise 13.5.2 answer

Once the loop decreases the size to one of the base cases, either the start number is positive or else the second number is.

```
[1]: def first_positive(numbers: list) -> int: # noqa: D103
    start = 0
    end = len(numbers)
    while end - start > 2:
        middle = start + (end - start) // 2
        if numbers[middle] > 0:
            end = middle + 1
        else:
            start = middle + 1
    if numbers[start] > 0:
        return numbers[start]
    else:
        return numbers[start + 1]
```

Exercise 13.5.3 answer

Since the numbers are in ascending order and unique, if the middle number is larger than its index, as happens in (1, 2, 3), then all numbers to its right are also larger than their indices. So we search the left half. Vice versa, if the middle number is below its index, as in (-1, 0, 2), so are the numbers to the left, so we search the right half.

Here's a recursive algorithm for `right_place(numbers, start, end)` with precondition $0 \leq start \leq end \leq |numbers|$:

1. if $end = start$:
 1. let $found$ be false
2. otherwise:
 1. let $middle$ be $start + \text{floor}((end - start) / 2)$
 2. if $numbers[middle] = middle$:
 1. let $found$ be true
 3. otherwise if $numbers[middle] < middle$:

1. let *found* be right_place(*numbers*, *middle* + 1, *end*)
4. otherwise:
 1. let *found* be right_place(*numbers*, *start*, *middle*)

Neither recursive search includes the middle number, so the slice is decreased.

30.11 Sorting

30.11.1 Insertion sort

Exercise 14.3.1 answer

If the in-place version is used, sorting a descending sequence takes linear time too. The in-place version changes the input sequence; so after the first `%timeit` loop of the first run sorts the descending sequence in quadratic time, all subsequent `%timeit` loops of all subsequent runs take linear time. In other words, only the first iteration of the first run is executed on the intended input, all other executions just measure how long it takes to sort an ascending sequence.



Note: If an algorithm modifies the input, measure each of its executions on a new copy of the input.

30.11.2 Selection sort

Exercise 14.4.1 answer

Step 1.2 takes constant time and hence can be ignored for analysis purposes. Like insertion sort, selection sort does $n - 1$ linear searches and hence is also quadratic. To be precise, step 1.1 first searches among n items, then among $n - 1$, then among $n - 2$, and so on until only searching two items. As we've seen before, $2 + \dots + n = (n^2 + n) / 2 - 1$.

Exercise 14.4.2 answer

SORTING—
SORGIN—T
NORGI—ST
NOIG—RST
NGI—ORST
IG—NORST
G—INORST

30.11.3 Merge sort

Exercise 14.5.1 answer

Slicing or merging each half takes $\Theta(n / 2)$ and sorting each half takes $T(n / 2)$, so the recursive definition is:

- if $n < 2$: $T(n) = \Theta(1)$
- if $n \geq 2$: $T(n) = 2 \times \Theta(n / 2) + 2 \times T(n / 2) + 2 \times \Theta(n / 2) = 2 \times T(n / 2) + \Theta(n)$.

This form of definition leads to $T(n) = \Theta(n \log n)$, confirming our informal analysis.

30.11.4 Quicksort

Exercise 14.6.1 answer

The pivot is the largest element, so again one partition is empty and the other has $n - 1$ items. The complexity is quadratic.

30.11.5 Quicksort variants

Exercise 14.7.1 answer

If all keys are the same, the sequence is sorted and two-way quicksort takes quadratic time, as seen in the previous section.

Three-way quicksort puts all items in the *equal* partition. The recursive calls on the other two partitions return immediately because they're empty. The algorithm takes linear time to create the partitions and concatenate them again.

Exercise 14.7.2 answer

The size of each partition varies from call to call, depending on how the pivot relates to the other items. quickselect decreases the input by a variable amount.

Exercise 14.7.3 answer

I use *item* as the output variable name.

A sequence can't be split if it's empty (which the preconditions don't allow) or has a single item, i.e. $n = 1$. In that case, it must be $k = 1$ and the smallest item is the only item in the sequence.

1. if $n = 1$:
 1. let *item* be *unsorted*[0]
2. otherwise:
 1. let (*smaller*, *pivot*, *larger*) be partition(*unsorted*)
 2. if $|\text{smaller}| = k - 1$:
 1. let *item* be *pivot*[0]
 3. otherwise if $|\text{smaller}| \geq k$:

1. let $item$ be quickselect($smaller, key, k$)
4. otherwise:
 1. let $item$ be quickselect($larger, key, k - |smaller| - 1$)

30.12 Rooted trees

30.12.1 Binary tree

Exercise 16.1.1 answer

1. All trees have size 7. Their heights are 3, 4 and 4, from left to right.
2. The nodes at level 2 are 3, 4, 5 and 6 in the left-hand tree, \times and 6 in the middle tree, and 3 and \times in the right-hand tree.
3. In the left-hand tree, the multiplication is a parent of the subtraction, so it's an ancestor, not a descendant. The multiplication is a child of the subtraction in the middle tree and a child of a child in the right-hand tree: in both cases it's a descendant.

Exercise 16.1.2 answer

```
[1]: %run -i ../m269_tree

MTP = join("-", join("*", join("+", THREE, FOUR), FIVE), SIX) #_
→ ((3+4)*5)-6
```

30.12.2 Algorithms on trees

Exercise 16.2.1 answer

The height is the number of levels, so we must see which subtree has most levels and add one for the extra level of the root.

- if $tree$ is empty: $\text{height}(tree) = 0$
- otherwise: $\text{height}(tree) = \max(\text{height}(\text{left}(tree)), \text{height}(\text{right}(tree))) + 1$

Exercise 16.2.2 answer

```
[1]: %run -i ../m269_tree
from algoesup import test

def height(tree: Tree) -> int:
    """Return the height of the tree."""
    if is_empty(tree):
        return 0
    else:
```

(continues on next page)

(continued from previous page)

```
    return max(height(tree.left), height(tree.right)) + 1

height_tests = [
    # case,           tree,     height
    ('empty tree',   Tree(),   0),
    ('(3+4)*(5-6)', TPM,     3),
    ('3+((4*5)-6)', PMT,     4),
    ('(3+(4*5))-6', MPT,     4),
]
test(height, height_tests)

Testing height...
Tests finished: 4 passed (100%), 0 failed.
```

30.12.3 Traversals

Exercise 16.3.1 answer

As mentioned earlier, the literals are the leaves of the expression trees. I add a base case to print leaves without enclosing brackets.

```
[1]: %run -i ./m269_tree

def infix(expression: Tree) -> None:
    """Print infix form of expression, with full brackets."""
    if is_empty(expression):
        return
    if is_leaf(expression):
        print(expression.root, end="")
        return
    print("(", end="")
    infix(expression.left)
    print(" ", expression.root, " ", end="")
    infix(expression.right)
    print(")", end="")
```

30.12.4 Binary search trees

Exercise 16.4.1 answer

The membership and lookup operations are very similar. The latter doesn't have a base case for the empty tree, because of the precondition, and returns the root's value, instead of a Boolean, for the other base case.

[1]: %run -i ./m269_tree

```
def lookup(tree: Tree, key: object) -> object:
    """Return the value associated to the key.

    Preconditions: tree is a BST and has the key
    """
    if tree.root[KEY] == key:
        return tree.root[VALUE]
    elif tree.root[KEY] < key:
        return lookup(tree.right, key)
    else:
        return lookup(tree.left, key)
```

Exercise 16.4.2 answer

An in-order traversal produces the desired outcome: it first visits the items with smaller keys in the left subtree, then the root and finally the items with larger keys in the right subtree.

Exercise 16.4.3 answer

The tree can't be empty because the smallest key is undefined in that case. The base case is when there's no left subtree: we've reached the left-most node. Otherwise the smallest key in the tree is the smallest in the left subtree. The tail recursive definition is

- if left(tree) is empty: smallest(tree) = root(tree)
- otherwise: smallest(tree) = smallest(left(tree)).

This is a tail-recursive definition. The iterative version is:

[1]: %run -i ./m269_tree

```
def smallest(tree: Tree) -> object:
    """Return the item in the tree with the smallest key."""
    while not is_empty(tree.left):
        tree = tree.left
    return tree.root
```

30.12.5 Balanced trees

Exercise 16.5.1 answer

1. False: a counter example is the right-hand tree in Figure 16.5.2. Every subtree is balanced but the overall tree isn't: the root has balance factor -2 .
2. True: if all nodes have the required balance factor, then so have the nodes in each subtree.

Exercise 16.5.2 answer

In the general case, each call takes

- $\Theta(1)$ to compute the height of an empty tree
- $\Theta(n - 1)$ to compute the height of the other subtree
- $T(0)$ to check the empty subtree is balanced
- $T(n - 1)$ to check the other subtree
- $\Theta(1)$ for the arithmetic and logical operations.

The base case takes constant time, so:

- if $n = 0$: $T(0) = \Theta(1)$
- if $n > 0$: $T(n) = \Theta(1) + \Theta(n - 1) + T(0) + T(n - 1) + \Theta(1) = T(n - 1) + \Theta(n)$.

This leads to $T(n) = \Theta(n^2)$.

This was to be expected: the function is computing the height for each node's single subtree and computing the height requires traversing the whole subtree. It does a linear operation n times, so it takes quadratic time.

Exercise 16.5.3 answer

If a node has an invalid balance factor, the tree is unbalanced so there's no point in checking whether the subtrees are. Likewise, if the left subtree is unbalanced, we don't need to check the right one. Using short-circuit conjunction, the code becomes simpler and faster for many trees:

```
if -1 <= height(tree.left) - height(tree.right) <= 1:  
    return is_balanced(tree.left) and is_balanced(tree.right)  
else:  
    return False
```

An even better approach for this problem is to use post-order traversal to avoid computing the height of a tall tree if a small subtree within it is unbalanced.

```
if not is_balanced(tree.left):  
    return False  
elif not is_balanced(tree.right):  
    return False  
else:  
    return -1 <= height(tree.left) - height(tree.right) <= 1
```

The code can be made shorter with a rather long short-circuit conjunction:

```
return is_balanced(tree.left) and is_balanced(tree.right) and -1 <= .  
→ ..
```

30.12.6 Heapsort

Exercise 16.6.1 answer

We can stop searching in a min-heap if the root is larger than the item sought, because all children must be even larger; but if the root is smaller than the item, we don't know if the item is in the left or right subtree. Searching in a heap takes linear time in the worst case.

Exercise 16.6.2 answer

As mentioned earlier, a complete tree has minimal height, which is $\log_2 n$, because each level is as full as possible. Each step (index arithmetic, swapping, comparing keys) takes constant time, so inserting one item and removing the root have logarithmic complexity. Heapsort is log-linear in the worst case.

Exercise 16.6.3 answer

If all keys are the same, there are no swaps on insertion and on removal, so each takes constant time. Heapsort takes linear time in such a scenario.

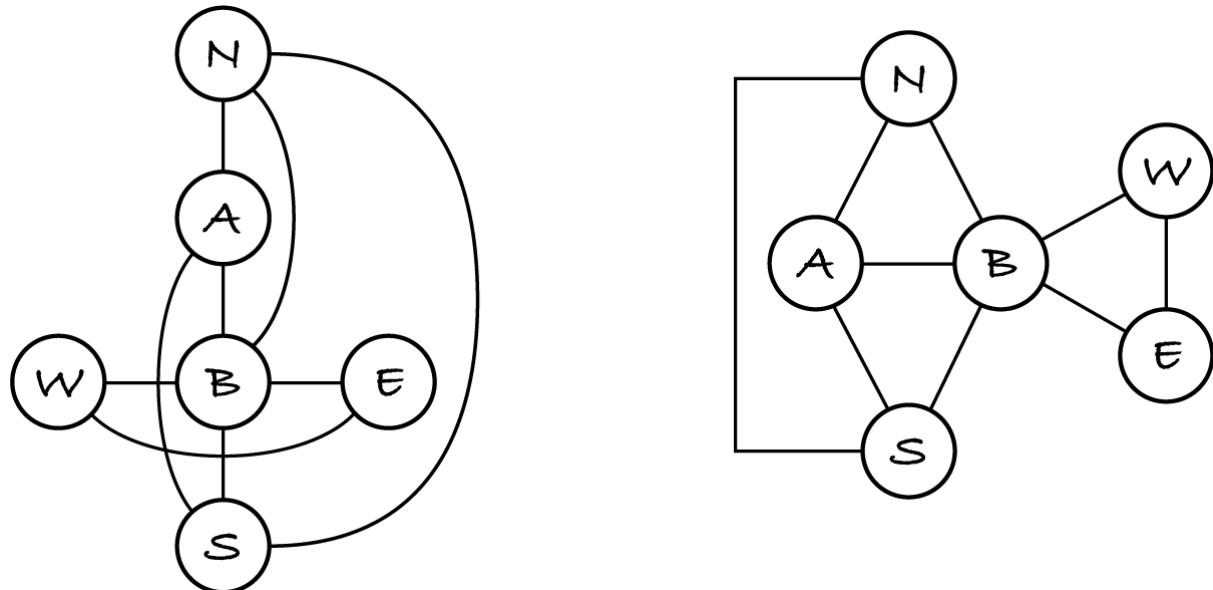
30.13 Graphs 1

30.13.1 Modelling with graphs

Exercise 17.1.1 answer

One can travel without changing trains between any two stations on the same line. The next figure shows the same graph drawn in two ways. The left layout obeys the geographical placement of the stations; the right layout avoids edges crossing.

Figure 17.1.5



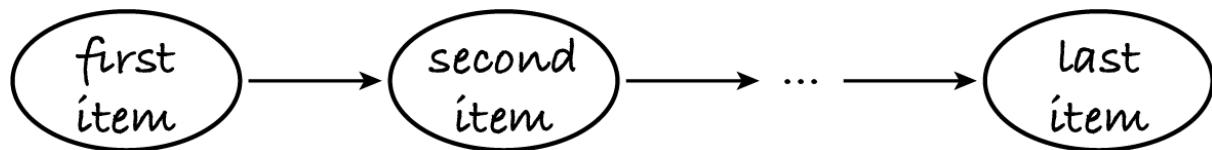
Exercise 17.1.2 answer

Being a sibling (or a cousin) is a symmetric relation but being a parent isn't: if A is a parent of B, then B is most certainly not a parent of A.

Exercise 17.1.3 answer

We can represent a sequence of items with a digraph that has one node per item. An edge from node A to node B represents the relation 'item A comes immediately before item B' in the sequence. The shape of the digraph is:

Figure 17.1.6



An alternative is for edges to be in the opposite direction, representing the relation 'comes immediately after'.

Exercise 17.1.4 answer

The set of items is represented by a graph with one node per item and no edges, since there is no relation between the items.

An alternative is to represent the relation 'is in the same set as', with undirected edges between all pairs of nodes.

Exercise 17.1.5 answer

I would use one graph for each rail or subway line. Each station would be a node in every graph representing a line that connects that station.

30.13.2 Basic concepts

Exercise 17.2.1 answer

I would select those nodes that have the highest degree because they correspond to the stations with most connections, hence likely to have the largest footfall (number of people arriving and departing).

Exercise 17.2.2 answer

1. In a null graph, all nodes have degree 0 because there are no edges.
2. In a path graph, the first and last nodes have degree 1; all other nodes have degree 2.
3. In a cycle graph, all nodes have degree 2.
4. In a complete graph, each node has degree $n - 1$ because it's adjacent to all other nodes.

Exercise 17.2.3 answer

1. A path graph has $n - 1$ edges.
2. A cycle graph has n edges.
3. A complete graph has $n \times (n - 1) / 2$ edges because each node connects to $n - 1$ other nodes, but each edge is being counted twice in that way.

Exercise 17.2.4 answer

1. Null graphs have no edges, so they are the sparsest among all graphs with the same number of nodes.
2. A path graph is a tree, so it's sparse.
3. A cycle graph has an edge/node ratio of one, so it's also sparse.
4. A complete graph has all possible edges, so it's the densest among all graphs with the same number of nodes.

30.13.3 Edge list representation

Exercise 17.3.1 answer

Nodes without neighbours (like node 3 in the example digraph) don't appear in the collection of edges, so we need a collection of nodes.

If we want to save memory, we could store only the nodes without neighbours, which will be very few in most real-world networks. However, that complicates the implementation of the graph operations.

Exercise 17.3.2 answer

Operation	Implementation	Complexity
add node A	add to the set of nodes	$\Theta(1)$
remove node A	remove A from the set of nodes and remove all pairs (A, B) and (B, A) from the set of edges	$\Theta(e)$
has edge (A, B)	check membership in the set of edges	$\Theta(1)$
add edge (A, B)	add to the set of edges	$\Theta(1)$
remove edge (A, B)	remove from the set of edges	$\Theta(1)$
in-neighbours of A	return all B such that (B, A) is in the set of edges	$\Theta(e)$
out-neighbours of A	return all B such that (A, B) is in the set of edges	$\Theta(e)$
in-degree of A	count edges of the form (B, A) in the set of edges	$\Theta(e)$

Exercise 17.3.3 answer

Yes it can. We simply add pair (A, A) to the collection of edges.

30.13.4 Adjacency matrix representation

Exercise 17.4.1 answer

Operation	Implementation	Complexity
has edge (A, B)	check if row A, column B is true	$\Theta(1)$
add edge (A, B)	set row A, column B to true	$\Theta(1)$
remove edge (A, B)	set row A, column B to false	$\Theta(1)$
in-neighbours of A	return all B such that row B, column A is true	$\Theta(n)$
out-neighbours of A	return all B such that row A, column B is true	$\Theta(n)$
in-degree of A	count the true values in column A	$\Theta(n)$

Exercise 17.4.2 answer

Most real-world graphs have more than one edge per node, so they have more edges than nodes. Operations that take $\Theta(e)$ on an edge list take $\Theta(n)$ on an adjacency matrix, so the latter is more efficient.

Exercise 17.4.3 answer

Yes, it can. We simply set row A, column A to true.

If there are no loops, as in the examples, the main diagonal is false.

30.13.5 Adjacency list representation

Exercise 17.5.1 answer

Operation	Implementation	Complexity
add node A	add key A with value [] to the dictionary	$\Theta(1)$
remove node A	remove A from all lists and delete the entry for A	$\Theta(e)$
has edge (A, B)	check if B is in the list for A	$\Theta(\text{out-degree}(A))$
add edge (A, B)	append B to the list for A	$\Theta(1)$
remove edge (A, B)	remove B from the list for A	$\Theta(\text{out-degree}(A))$
in-neighbours of A	return all B such that A is in the list for B	$\Theta(e)$
out-neighbours of A	return a copy of A's out-neighbours	$\Theta(\text{out-degree}(A))$
in-degree of A	count how many lists include A	$\Theta(e)$

You may have considered the out-neighbours operation to take constant time.

Exercise 17.5.2 answer

Operation	Complexity
add node A	$\Theta(1)$
remove node A	$\Theta(n)$
has edge (A, B)	$\Theta(1)$
add edge (A, B)	$\Theta(1)$
remove edge (A, B)	$\Theta(1)$
in-neighbours of A	$\Theta(n)$
out-neighbours of A	$\Theta(\text{out-degree}(A))$
in-degree of A	$\Theta(n)$

Exercise 17.5.3 answer

We simply include A in the adjacency list of A.

30.13.6 Classes for graphs

Exercise 17.6.3 answer

The following solution collects all degree values and checks there's only one. A more efficient algorithm would stop as soon as a different degree is found, without using a set. I'll leave that as an optional exercise.

```
[1]: %run -i ../m269_digraph
%run -i ../m269_ungraph

def same_degree(graph: UndirectedGraph) -> bool:
    """Return True if and only if all nodes have the same degree.

    Preconditions: graph isn't empty
    """
    all_degrees = set()
    for node in graph.nodes():
        all_degrees.add(graph.degree(node))
    return len(all_degrees) == 1
```

30.13.7 Traversing a graph

Exercise 17.7.1 answer

The input graph is connected if and only if all nodes are mutually reachable. No matter which start node we pick, all others can be reached.

The algorithm is: Pick any node as the start node. Do a traversal from it. Return true if and only if the returned sequence has as many nodes as the input graph.

Exercise 17.7.2 answer

The simplest algorithm is to first check if $n - 1 = e$ for the input graph. If it isn't, the graph can't be a tree. If it is, use the algorithm from the previous exercise to check if the input graph is connected.

An alternative, less efficient approach is based on the following reasoning. If the input is a tree, there's only one path between each pair of nodes. The traversal therefore has no choice in how to reach the other nodes from the start. The traversed subgraph will be the whole tree.

The algorithm does a traversal from some start node and then checks if every node and edge of the input graph is also in the generated subgraph. Since the input graph is undirected but the output graph is directed, we must check for each edge (A, B) of the input if (A, B) or (B, A) are in the output.

30.14 Greed

30.14.1 Interval scheduling

Exercise 18.1.1 answer

Consider a knapsack with a capacity of 3 kg and two items: one is worth £3 and weighs 3 kg, the other is worth £2 and weighs 2 kg. The value is maximised by packing the £3 item, which fills the knapsack.

1. Choosing the lightest first will put only the £2 item in the knapsack.
2. The items are equally profitable (1 £/kg). Since there's a tie for the choice, the algorithm may make the wrong one, choosing the £2 item.

Exercise 18.1.2 answer

Step 2 has complexity $\Theta(i \log i)$. The loop is executed i times, but all its steps take constant time. The worst-case complexity is $\Theta(i \log i) + \Theta(i) = \Theta(i \log i)$.

30.14.2 Minimum spanning tree

Exercise 18.3.1 answer

I changed some variable names to be more meaningful.

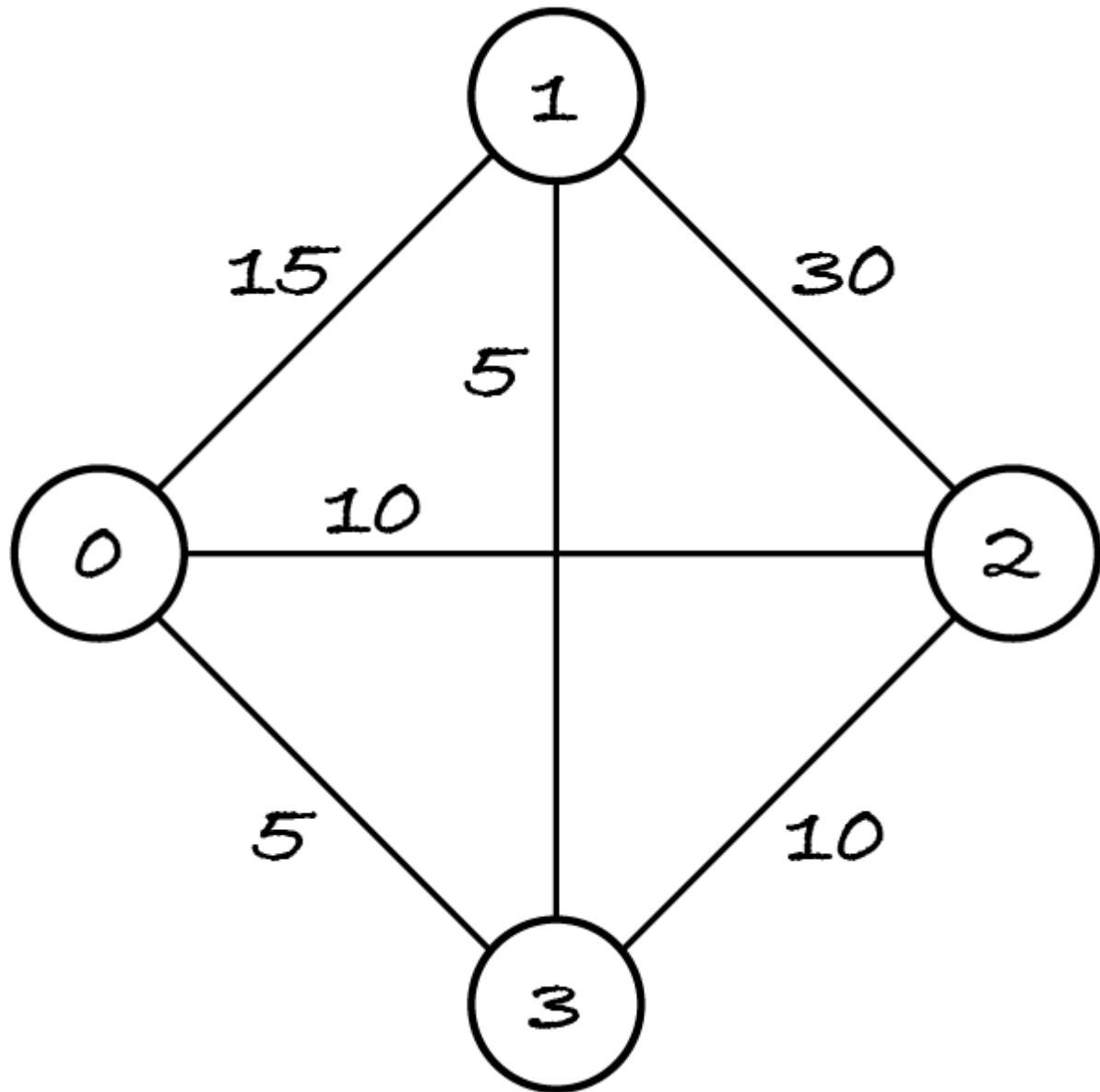
1. let $tour$ be the weighted undirected graph with single node $start$
2. let $current$ be $start$
3. repeat $n - 1$ times:
 1. find an edge $(current, next)$ with lowest weight($current, next$) such that $next$ isn't in $tour$
 2. add node $next$ to $tour$
 3. add edge $(current, next, weight(current, next))$ to $tour$

4. let *current* be *next*
4. add edge (*current*, *start*, weight(*current*, *start*)) to *tour*

Exercise 18.3.2 answer

The following graph has two best tours starting at 0 with total cost 40, as I remarked *earlier*: (0, 1, 3, 2, 0) and (0, 2, 3, 1, 0).

Figure 18.3.3



However, the greedy algorithm constructs tour (0, 3, 1, 2, 0) with cost 50 because node 3 is nearest to node 0, and node 1 is nearest to node 3.

Exercise 18.3.3 answer

Previous traversal algorithms use sets, stacks and queues for the *unprocessed* collection, with edges added and removed in constant time. With a priority queue, adding and removing edges takes logarithmic time. One way to make the algorithm more efficient is to not make the queue longer than necessary: when adding the edges of the just-visited node, we should only add those that lead to yet-unvisited nodes.

Even though this change puts only edges to unvisited nodes in the priority queue, the algorithm must still check in step 4.3 if the edge removed from the priority queue leads to an unvisited node. I'll leave it as an exercise for you to explain to your study buddy or in the forums why that's the case.

Exercise 18.3.4 answer

If all edges have the same weight w then every spanning tree has the same total weight $(n - 1) \times w$, i.e. any spanning tree is an MST.

We can thus ignore the weights and use any traversal algorithm, because it produces a spanning tree if the input is connected. DFS produces a 'long and thin' spanning tree, while BFS produces a 'bushy' tree.

30.14.3 Shortest paths

Exercise 18.4.1 answer

If all weights are the same, a shortest path has the fewest edges, so it suffices to use BFS.

Exercise 18.4.2 answer

The algorithm can stop constructing the tree after visiting the end node, e.g. by emptying the queue. If the tree construction ends normally, without ever visiting the end node, then the algorithm returns the empty sequence and stops.

Otherwise the algorithm proceeds to obtain the path from the tree. In a graph that represents a rooted tree, each node except the root has a single in-neighbour. The algorithm can thus traverse the path in reverse by visiting the in-neighbour of the end node, then the in-neighbour of that node, and so on, until getting to the start node, appending each node to a sequence as it visits them. Finally, the algorithm reverses the sequence and returns it.

Exercise 18.4.3 answer

I wish to capture the intuition that a central station is in the middle of all other stations. One can more quickly reach all other stations than from a peripheral station.

Let lc be the largest cost of all shortest paths starting at a node A. In other words, all nodes can be reached from A with cost lc or less. I consider a node to be central if it has the smallest lc value of all nodes.

To compute the central nodes, I do a linear search over all nodes, computing the lc of each node and keeping the lowest value found so far and the set of nodes which have it.

To compute the lc for a node, I run Dijkstra's algorithm from that node, but instead of returning a tree I return the cost of the last node visited, i.e. the priority of the last edge added to the tree.

If you have come up with a different notion of central node, please share it in the forums.

30.15 Practice 2

30.15.1 Jousting

Exercise 19.1.1 answer

To simulate the tournament, we use a priority queue where the items are the knights' indices and the associated priorities are their strengths, which change over the tournament. The algorithm is as follows, where *strength* is the input sequence.

For each index k , add k with priority $strength[k]$ to a max-priority queue. While the queue has length larger than one, remove the two front items (indices). If the first has a larger priority (strength) than the second, put the first back in the queue but with the reduced strength as the new priority. (If both priorities are the same, both items have been removed from the queue.)

If the loop ends with an empty queue, the output is -1 , otherwise it's the last remaining index in the queue.

Exercise 19.1.2 answer

In the worst-case scenario, each joust eliminates one, not two, knights, so there are $n - 1$ jousts, until one knight remains.

Implementing the priority queue with a heap, adding or removing one knight to the queue takes logarithmic time. First, we add n knights to an initially empty queue: this takes log-linear time. Then, we remove two knights and add one back for each joust. The three operations take logarithmic time. Doing this $n - 1$ times takes log-linear time.

The worst-case complexity of the whole algorithm is log-linear in the number of knights.

Exercise 19.1.3 answer

```
[1]: from algoesup import test
      from heapq import heappush, heappop

      def winner(strength: list) -> int:
          """Return the index of the winning knight or -1 if no one wins.

          Preconditions: all items in strength are positive integers
          """
          remaining = [] # priority queue of knights still jousting
          for knight in range(len(strength)):
              heappush(remaining, (-strength[knight], knight))

          while len(remaining) > 1:
              first = heappop(remaining)
              second = heappop(remaining)
              if first[0] < second[0]:
                  heappush(remaining, (first[0], first[1]))
```

(continues on next page)

(continued from previous page)

```

while len(remaining) > 1:
    # take the two strongest knights
    first = heappop(remaining)
    second = heappop(remaining)
    # if they don't knock each other out
    if first[0] != second[0]:
        # the first (stronger) remains with diminished strength
        heappush(remaining, (first[0] - second[0], first[1]))
if len(remaining) == 0:
    return -1
else:
    return remaining[0][1] # return knight at head of queue

winner_tests = [
    # case,           strength,   winner
    ('no one wins', [5, 3, 6, 2], -1),
    ('second wins', [5, 3, 6, 1],  1), # winner has initial
→strength 3
    # your tests
    ('no jousts',   [5],         0),
    ('no knights',  [],          -1),
    # if all knights have the same strength, no one or last one wins
    ('last wins',   [1, 1, 1],    2),    # odd number: last wins
    ('all KO',      [1, 1, 1, 1], -1)    # even number: all KO
]
test(winner, winner_tests)

Testing winner...
Tests finished: 6 passed (100%), 0 failed.

```



Info: This is a modification of LeetCode problem [1046](#).

30.15.2 Dot product

Exercise 19.2.1 answer

1. For each permutation of Y , compute its dot product with X and keep it if it's lower than the previous results. Keeping X as it is and rearranging only Y will multiply each integer in X with each integer in Y and thus generate all dot products.
2. There are $n!$ permutations of Y to go through. Each permutation takes $\Theta(n)$ time to test if it's the best one, because the dot product requires n multiplications and n additions. The complexity is $\Theta(n! \times n)$.

3. For $n = 20$, if each candidate is generated and tested in one nanosecond, we'd have to wait...

```
[1]: from math import factorial

NS_PER_YEAR = 365 * 24 * 60 * 60 * 1000 * 1000 * 1000 # nanoseconds in a year
print(factorial(20) * 20 // NS_PER_YEAR, "years")

1542 years
```

Exhaustive search is not a practical approach for this problem.

Exercise 19.2.2 answer

To minimise the contribution of the highest value in Y to the dot product, I multiply it with the lowest value in X . My greedy choice at each step is to take the lowest unprocessed value in X and the largest unprocessed value in Y , multiply them and add the product to the sum so far.

An alternative is to choose at each step the largest unprocessed value in X and the lowest unprocessed value in Y .

Both greedy choices pair the largest, second largest, etc. value of one sequence with the smallest, second smallest, etc. value of the other. The example rearrangements do exactly that, so these greedy choices work for the examples given.

If you found a different greedy choice that produces the same dot product for the example inputs, please share it in the M269 forums.

Exercise 19.2.3 answer

Since I take at each step the lowest of the X and the highest of the Y , I simply sort the two sequences in ascending and descending order, respectively. I don't sort them in-place because I don't want to modify the inputs.

```
[1]: from algoesup import test

def min_dot_product(x: list, y: list) -> int:
    """Return the smallest dot product over all permutations of x and y.

    Preconditions:
    x and y are non-empty lists of integers of the same length
    """
    x_up = sorted(x)
    y_down = sorted(y, reverse=True)
    product = 0
    for index in range(len(x)):
        product = product + x_up[index] * y_down[index]
```

(continues on next page)

(continued from previous page)

```
return product

min_dot_product_tests = [
    # case,           x,           y,           smallest
    ('n = 3',        [1, 2, -1],   [2, 6, 3],   1),
    ('n = 5',        [1, -1, 5, 0, 7], [3, 7, -9, 2, 0], -68),
    # your tests
    ('n = 1',        [4],         [-3],        -12),
    ('x same',       [1, 1, 1],   [2, 4, 3],   9),
    ('x up, y up',  [1, 2, 3, 4], [-1, 0, 5, 7], 13),
]

test(min_dot_product, min_dot_product_tests)

Testing min_dot_product...
Tests finished: 5 passed (100%), 0 failed.
```



Info: These exercises are based on problem [Minimum Scalar Product](#).

30.15.3 Beams

Exercise 19.3.1 answer

The smallest input (empty set: no beams) leads to the smallest output (empty set: no weak points). There should also be tests for unconnected beams without common points and for a normal structure, with some strong and some weak points.

Exercise 19.3.2 answer

Being joined by a beam is a symmetric relation between points, so the structure is modelled as an undirected unweighted graph, with points as nodes and beams as edges.

Exercise 19.3.3 answer

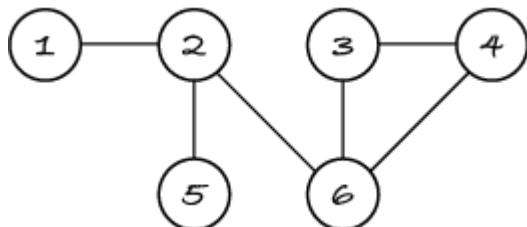
An exhaustive search over the graph finds all weak nodes. For each candidate node, if it's not part of a triangle, add the node to an initially empty set of weak nodes. At the end of the search, that set is the output.

To test if a candidate node A isn't in a triangle, do another brute-force search: generate all pairs of neighbours B and C and check if there's an edge (B, C). If there is, stop the search: edges (A, B), (A, C) and (B, C) form a triangle and therefore nodes A, B and C are strong. If the search ends without finding any edge (B, C), then A is weak.

For the example graph given in the hint and repeated below, when testing if node 2 is weak, the algorithm generates all pairs of neighbours (1, 5), (1, 6) and (5, 6) and checks if any of them

forms an edge. None does, so node 2 is weak. When testing if node 6 is weak, the algorithm generates the pairs of neighbours (2, 3), (2, 4) and (3, 4) but the latter is an edge, so node 6 is strong, and so are nodes 3 and 4.

Figure 19.3.2



An alternative algorithm first finds all strong points and finally returns the weak points as the difference between all points and the strong points.

Exercise 19.3.4 answer

The input is a set of edges, so each node in the input has neighbours.

The search does the least work when no node has a pair of neighbours, i.e. each node has exactly one neighbour and thus no two beams have a common point, as in the next example.

Figure 19.3.3



In that case, the test of whether a node is in a triangle takes constant time, because the candidate has no pairs of neighbours. The algorithm tests the n nodes and adds all to the set of weak nodes. The best-case complexity is $\Theta(n)$.

The search does the most work when each node has all other ones as neighbours, i.e. the graph is complete. The algorithm obtains the $n - 1$ neighbours, but detects in constant time that the first pair is connected and so the node isn't weak. Obtaining all neighbours of all nodes takes $\Theta(n^2)$ time. None of the nodes is added to the set of weak nodes. The worst-case complexity is quadratic in the number of nodes.

Exercise 19.3.5 answer

I use an *auxiliary nested function* to test each node.

As for the tests, the best case (Exercise 19.3.4) also covers the case of beams without common points (Exercise 19.3.1).

```
[1]: %run -i ../m269_digraph
%run -i ../m269_ungraph
%run -i ../m269_graphs
from algoesup import test

def weak_points(beams: set) -> set:
```

(continues on next page)

(continued from previous page)

```
"""Return the points that aren't part of any triangle.
```

*beams is a set of pairs of integers.
The output is a set of integers.
Each integer represents a point.*

*Preconditions: for every pair (a, b), a ≠ b
Postconditions: the output only has points occurring in beams*

```
"""
```

```
def is_weak(point: int, structure: UndirectedGraph) -> bool:
    """Return True if point isn't part of a triangle in
    →structure."""
    neighbours = structure.neighbours(point)
    for point2 in neighbours:
        for point3 in neighbours:
            if structure.has_edge(point2, point3):
                return False
    return True

structure = UndirectedGraph()
for beam in beams:
    if not structure.has_node(beam[0]):
        structure.add_node(beam[0])
    if not structure.has_node(beam[1]):
        structure.add_node(beam[1])
    structure.add_edge(beam[0], beam[1])

weak = set()
for point in structure.nodes():
    if is_weak(point, structure):
        weak.add(point)
return weak

T134 = {(1, 3), (1, 4), (3, 4)} # some triangles
T346 = {(3, 4), (4, 6), (6, 3)}
worst = complete_graph(5) # complete graphs are a worst case

weak_points_tests = [
    # case,           beams,           weak points
    ('no triangle',   {(1,2), (3,1)}, {1, 2, 3}),
    ('missing points', {(7,3), (3,2)}, {2, 3, 7}),
    # your tests:
    ('no points',     {},             set()),
    ('mixed points', T134|T346|{(2,1), (6,9)}, {2, 9}),
]
```

(continues on next page)

(continued from previous page)

```

('best case',           { (1, 3), (5, 2) },           { 1, 2, 3, 5 } ),
('worst case',         worst.edges(),             set() )

]

test(weak_points, weak_points_tests)
Testing weak_points...
Tests finished: 6 passed (100%), 0 failed.
    
```

With a small change, the algorithm becomes more efficient, but without improving the worst-case complexity. You may wish to discuss this with your study buddy or in the forums. (Hint: do you need to always check all neighbours?)



Info: These exercises are based on problem [Weak Vertices](#).

30.15.4 Up and down

Exercise 19.4.1 answer

We can adapt binary search because each part of the sequence is sorted. This will lead to a very efficient logarithmic algorithm.

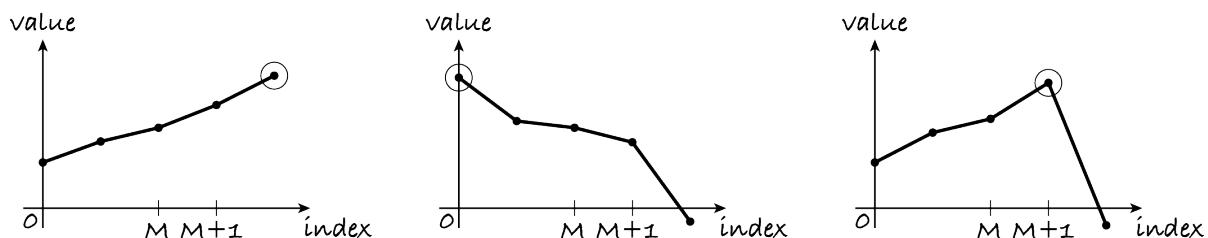
Exercise 19.4.2 answer

The base case is when the sequence has one number, which must be the largest.

To know in which part the middle integer is, we must compare it to its left or right neighbour. Let's use the number to the right of the middle.

The next figure shows the same sequence shapes with five values as before. The x -axis includes indices M and $M+1$ for the middle value and next value.

Figure 19.4.2



If the middle integer is larger than the integer to its right, as it happens in the central chart, then the middle integer is in the descending part, so the maximum is in the middle (if it's the start of the descending part) or in the left half.

There are no repeated consecutive integers, so if the middle integer isn't larger than the next integer, it must be smaller, as it happens in the left- and right-hand charts. This means the middle integer is in the ascending part, so the maximum is in the right half, excluding the middle.

So far I'm assuming there's a value after the middle one. What if there's none, i.e. if the middle number is the last value of the sequence? This only happens for sequences of length 1 (already treated as a base case) and 2.

If there are only $n = 2$ numbers in the sequence, then the middle number is at index $n / 2 = 1$. The middle number is the second one and there's no number to the right to compare it to. One simple solution is to handle sequences of two numbers as another base case and return the larger of both numbers.

Exercise 19.4.3 answer

```
[1]: from algoesup import test

def max_up_down(numbers: list) -> int:
    """Return the largest of numbers.

    Preconditions:
    - numbers is a non-empty list of non-duplicated integers
    - numbers first ascend and then descend
    - the ascending or the descending part may be empty
    """
    start = 0
    end = len(numbers)  # search in the slice [start:end]
    while end - start > 2:
        middle = start + (end - start) // 2
        # at least 3 numbers, so there's one after the middle one
        if numbers[middle] < numbers[middle + 1]:
            # ascending part: maximum comes after the middle
            start = middle + 1
        else:
            # descending part: maximum is middle or to its left
            end = middle + 1
    if end - start == 2:
        return max(numbers[start], numbers[start + 1])
    else:
        return numbers[start]

max_up_down_tests = [
    # case,           numbers,          largest
    ('1 number',      [-1],             -1),
    ('only descending', [3, 2],           3),
    ('only ascending', [-3, -1, 0],       0),
    ('up and down',   [-3, -1, 0, 3, 2, -1], 3),
]
test(max_up_down, max_up_down_tests)
```

```
Testing max_up_down...
Tests finished: 4 passed (100%), 0 failed.
```



Info: This is LeetCode problem [852](#).

30.15.5 A knight goes places

Exercise 19.5.1 answer

An edge case is the smallest board, with one row and one column. In that case, the start and end square must be the same.

The smallest boards for which the start and end squares can differ are the 1×2 and 2×1 boards. The knight can't move in either of them. Both should be tested with one square being the start and the other the end.

The start and end squares being the same is another edge case. This should be tested with a larger board, e.g. 3×3 , in which the knight has space to move.

Exercise 19.5.2 answer

If the knight can move from square A to square B, it can also move back to A. In other words, the relation 'can move from A to B' is symmetric. I'll use an undirected unweighted graph: the nodes represent the squares and an edge connects two nodes if it's possible for the knight to move from one to the other.

Exercise 19.5.3 answer

A breadth-first search will find the shortest path from the start to the end square, if there is one. The algorithm in [Section 17.8](#) must be modified. It must stop as soon as the end node is visited. It must otherwise return -1, i.e. when the queue gets empty. Moreover, the required output is the length of the shortest path, not the traversed tree. So when adding an edge to the queue we should include the number of the move that corresponds to following that edge.

Here's the outline of the whole algorithm. I use numbered lists and sublists to make it easier to follow.

1. If the end square is the start square:
 1. Output zero and stop.
2. Otherwise:
 1. Create a graph with one node per square, and one edge between A and B if squares A and B are one knight move apart.
 2. Create a set of already visited nodes, initially just with the start square.
 3. Create a queue with triples (S, A, 1) where S is the start square and A is a neighbour (in the graph, not in the board) of S.

4. While the queue isn't empty, do the following:
 1. Remove the front triple (A, B, L) from the queue.
 2. If B is the end square:
 1. Output L and stop.
 3. If B hasn't been visited:
 1. Add B to the visited nodes.
 2. Add to the queue all triples (B, C, L+1) where C is a neighbour of B.
 5. Output -1 because the end node wasn't reached.

Exercise 19.5.4 answer

```
[1]: from algoesup import test

%run -i ../m269_queue
%run -i ../m269_digraph
%run -i ../m269_ungraph

def knight_moves(size: tuple, start: tuple, end: tuple) -> int:
    """Return least number of knight moves from start to end.

    Return -1 if end is not reachable from start.

    Preconditions:
    - size is a pair (rows, columns) with rows > 0 and columns > 0
    - start and end are pairs (r, c) with 0 <= r < rows and 0 <= c < columns
    """
    ...

def can_move(square1: tuple, square2: tuple) -> bool:
    """Check if a knight can move from one square to the other.

    Preconditions:
    the inputs are pairs (r, c) with 0 <= r < rows and 0 <= c < columns
    """
    vertical = abs(square1[0] - square2[0]) # movement between rows
    horizontal = abs(square1[1] - square2[1]) # movement between columns
    movement = (vertical, horizontal)
    return movement == (2, 1) or movement == (1, 2)

if end == start:
```

(continues on next page)

(continued from previous page)

```

        return 0

graph = UndirectedGraph()
for row in range(size[0]):
    for column in range(size[1]):
        graph.add_node((row, column))
for node1 in graph.nodes():
    for node2 in graph.nodes():
        if can_move(node1, node2):
            graph.add_edge(node1, node2)

visited = {start}
unprocessed = Queue()
for neighbour in graph.out_neighbours(start):
    unprocessed.enqueue((start, neighbour, 1))
while unprocessed.size() > 0:
    edge = unprocessed.dequeue()
    previous = edge[0]
    current = edge[1]
    length = edge[2]
    if current == end:
        return length
    elif current not in visited:
        visited.add(current)
        for neighbour in graph.out_neighbours(current):
            unprocessed.enqueue((current, neighbour, length + 1))
return -1

knight_moves_tests = [
    # case,           size,   start,   end,      moves
    ('1x1 board',     (1, 1), (0, 0), (0, 0),      0),
    ('1 row, 2 cols', (1, 2), (0, 0), (0, 1),     -1),
    ('2 rows, 1 col', (2, 1), (1, 0), (0, 0),     -1),
    ('start = end',   (3, 3), (1, 1), (1, 1),      0),
    ('bottom left',   (5, 6), (3, 4), (4, 0),      3), # figure 19.
    ↪5.1
    ('bottom right',  (5, 6), (3, 4), (4, 5),      4), # figure 19.
    ↪5.1
    ('3x3 to centre', (3, 3), (0, 0), (1, 1),     -1), # figure 19.
    ↪5.2
]

test(knight_moves, knight_moves_tests)

Testing knight_moves...
Tests finished: 7 passed (100%), 0 failed.

```



Info: These exercises are based on problem Gregory the Grasshopper.

30.16 Graphs 2

30.16.1 Undirected graph components

Exercise 21.1.1 answer

One algorithm can be outlined as follows:

Create a set of disconnected nodes, initially with all the graph's nodes. Compute the connected components. For each node V and for each source node S, if V and S are in the same component, then V is connected to the power grid, so remove V from the set. Return the remaining set after the loops finish.

A more efficient algorithm computes only the components of the source nodes, by traversing the graph from each source. All nodes not in those components are disconnected from the power grid. We don't need to know the individual components, so we don't need a map. We can merge the source node components into a single set of reachable nodes.

In the following code I use BFS instead of DFS to emphasise that any traversal can be used to find components.

```
[1]: %run -i ../m269_digraph
%run -i ../m269_ungraph
%run -i ../m269_queue
%run -i ../m269_stack

def disconnected(graph: UndirectedGraph, sources: set) -> set:
    """Return all nodes not connected to any of the sources.

    Preconditions: sources is a non-empty subset of the graph's nodes
    """
    connected = set()
    for source in sources:
        if source not in connected:
            reached = bfs(graph, source).nodes()
            connected = connected.union(reached)
    return graph.nodes() - connected
```

This bespoke algorithm draws on the two main ideas for computing components:

- a traversal from node A finds the nodes in the same component as A
- if we know the component of node A, we don't do a traversal from it.

Exercise 21.1.2 answer

Here's my proposal.

We're told the graph is connected; therefore it has one component. Removing a node and its edges may split the graph into two or more components. The more components, the more 'disconnection' the removal causes. The critical nodes would be those leading to the largest number of components, after the node's removal.

If you have a different definition, please share it in the forums.

30.16.2 Directed graph components

Exercise 21.2.1 answer

```
[1]: %run -i ../m269_digraph.py
%run -i ../m269_ungraph.py
%run -i ../m269_queue
%run -i ../m269_stack

def weakly_connected_components(graph: DiGraph) -> dict:
    """Return the weakly connected components of graph.

    Postconditions: the output maps each node to its component,
    numbered from 1 onwards.
    """
    undirected = UndirectedGraph()
    for node in graph.nodes():
        undirected.add_node(node)
    for edge in graph.edges():
        undirected.add_edge(edge[0], edge[1])
    return connected_components(undirected)
```

30.16.3 Topological sort

Exercise 21.3.1 answer

Method `in_degree(A)` goes through all nodes to check if `A` is each node's out-neighbour. This takes $\Theta(n)$. The complexity of Alice's loop is $n \times (\Theta(n) + \Theta(1)) = \Theta(n^2)$.

By contrast, the earlier version of Kahn's algorithm takes $\Theta(n+e)$ to initialise `indegree` and `to_visit`. Since e is much less than n^2 for most graphs, the original code is more efficient.

Exercise 21.3.2 answer

In a cyclic digraph a group of nodes depend on each other. Their in-degrees are never decremented to zero and none of them is ever visited. At some point the algorithm stops without having further nodes to visit, but the output sequence doesn't have all the digraph's nodes.

Exercise 21.3.3 answer

1. I just need to check the sequence doesn't have all the n nodes of the digraph.

```
[1]: %run -i ../m269_digraph
```

```
def is_cyclic(graph: DiGraph) -> bool:  
    """Return True if and only if the graph has a cycle."""  
    return len(topological_sort(graph)) < len(graph.nodes())
```

The expression

```
set(topological_sort(graph)) != graph.nodes()
```

would be correct too, but would take longer and use more memory because of the conversion to a set.

2. Obtaining the graph's nodes always takes $\Theta(n)$. Computing and comparing the sizes of the sequence and the set always takes $\Theta(1)$. Kahn's algorithm takes $\Theta(n + e)$ when the digraph is acyclic. If it's cyclic, the construction of the topological sort ends early. So the worst-case scenario is an acyclic digraph and the worst-case complexity is $\Theta(n + e) + \Theta(n) + \Theta(1) = \Theta(n + e)$.

Exercise 21.3.4 answer

1. For a cycle to exist there must be at least two nodes A and B such that there is a path from A to B and from B to A. But that's the same as saying that A and B are mutually reachable and therefore in the same strongly connected component. Thus, if there's a cycle there must be a component with two or more nodes, and vice versa, if there's a component with two or more nodes, that component has a cycle.
2. The worst-case scenario for Bob's algorithm is to go through the whole graph without finding a component with two or more nodes, i.e. the input is acyclic. Bob's algorithm has the same worst-case complexity as the original algorithm for strongly connected components, $\Theta(n \times (n+e))$, which isn't more efficient than using Kahn's.

30.16.4 State graphs

Exercise 21.4.1 answer

1. There's one node per square and for each of the three move distances, so $n = 3rc$.
2. Each node has at most four edges, because at most one move is possible in each direction in each state, so $e \leq 4n = 12rc$.
3. The complexity is thus $\Theta(n + e) = \Theta(3rc + 12rc) = \Theta(rc)$.

30.16.5 Practice

Exercise 21.5.1 answer

Islands are separated by water from each other. If we see the image as a graph, each island is a connected component of land. The most similar problem is finding the connected components of a graph.

Exercise 21.5.2 answer

We can find connected components in undirected graphs, and weakly and strongly connected components in directed graphs. The graphs may be weighted, but the weights are ignored.

To determine the islands we only need to know which land squares are vertically and horizontally adjacent to each other. There's no need for directed paths. Or put differently, the adjacency relation is symmetric. This means an undirected graph suffices. We need the algorithm for finding the connected components of an undirected graph.

Exercise 21.5.3 answer

We construct an unweighted, undirected graph from the input, with one node per land square and one edge A–B for each pair of vertically or horizontally adjacent squares A and B.

Exercise 21.5.4 answer

The algorithm that finds the connected components keeps a component counter and adds the nodes reached in each partial traversal to a map. There are two alternative ways to obtain the number of islands:

- Change the algorithm to only keep the counter and not use a map. When the algorithm stops, return the counter.
- Apply the algorithm unchanged. Then do a linear search over the returned map to find the highest component number.

30.17 Backtracking

30.17.1 Back to the TSP

Exercise 22.5.1 answer

I changed the function header to reflect that candidates are paths and that the value of a path is its length.

```
def length(path: list, graph: WeightedUndirectedGraph) -> int:  
    """Return the length of the path in graph."""  
    total = 0  
    for index in range(1, len(path)):  
        total = total + graph.weight(path[index-1], path[index])  
    return total
```

(continues on next page)

(continued from previous page)

```
length([0, 1, 2, 3, 4, 0], example) == 55
```

Exercise 22.5.2 answer

To check if a path ends with node 0, we don't need the instance graph so we can omit the `instance` parameter of the template.

I renamed the function to better reflect that the global constraint is for the candidate path to be a tour, i.e. to return to node 0.

```
def is_tour(path: list) -> bool:  
    """Check if the path ends at the start node."""  
    return path[-1] == 0
```

Exercise 22.5.3 answer

```
from typing import Hashable  
  
def can_extend(node: Hashable, path: list, graph:  
    ↪WeightedUndirectedGraph, best: list) -> bool:  
    """Check if node can extend path towards a shorter tour."""  
    return graph.has_edge(path[-1], node) and length(path+[node],  
    ↪graph) < best[VALUE]
```

Exercise 22.5.4 answer

I replace the template functions `satisfies_global` and `value` with `is_tour` (Exercise 22.5.2) and `length` (Exercise 22.5.1), respectively. The `can_extend` function was implemented in Exercise 22.5.3.

```
def extend(path: list, nodes: set,  
          graph: WeightedUndirectedGraph, shortest: list) -> None:  
    """Update shortest if path is a shorter tour, otherwise extend  
    it."""  
    print('Visiting node', path, nodes)  
    if len(nodes) == 0:  
        if is_tour(path):  
            path_length = length(path, graph)  
            if path_length < shortest[VALUE]:  
                print('New shortest tour with length', path_length)  
                shortest[SOLUTION] = path  
                shortest[VALUE] = path_length  
    for node in nodes:  
        if can_extend(node, path, graph, shortest):  
            extend(path + [node], nodes - {node}, graph, shortest)
```

Exercise 22.5.5 answer

Here is the modified code, without the docstring and the print statements.

```
def extend(path: list, nodes: set,
           graph: WeightedUndirectedGraph, shortest: list) -> None:
    if len(nodes) == 0:
        # if is_tour(path):                      # removed line
        path_length = length(path, graph)
        if path_length < shortest[VALUE]:
            shortest[SOLUTION] = path
            shortest[VALUE] = path_length
    for node in nodes:
        if node != 0 or nodes == {0}:      # added line
            if can_extend(node, path, graph, shortest):
                extend(path + [node], nodes - {node}, graph,_
                         shortest)
```

30.17.2 Back to the knapsack

Exercise 22.7.1 answer

A candidate is the new best solution if it has a higher value or if it has the same value but fewer items. The first if-statement in `extend` becomes:

```
if candidate_value > best[VALUE] or (candidate_value == best[VALUE]_-
                                         and len(candidate) < len(best[SOLUTION])):
```

30.18 Dynamic Programming

30.18.1 Longest common subsequence

Exercise 23.2.1 answer

[1]: %run -i ./m269_rec_list

```
def lcs(left: str, right: str) -> str:
    """Return the longest common subsequence of both strings."""
    if left == '' or right == '':
        return ''
    elif head(left) == head(right):
        return head(left) + lcs(tail(left), tail(right))
    else:
        skip_right = lcs(left, tail(right))
        skip_left = lcs(tail(left), right)
        if len(skip_left) > len(skip_right):
            return skip_left
        else:
            return skip_right
```

(continues on next page)

(continued from previous page)

```

        return skip_left
else:
    return skip_right

```

When skipping the left or right head leads to equally long common subsequences, I choose to skip the right head. Writing \geq instead of $>$ would skip the left head instead in such cases. The following cell produces a different LCS depending on which head is skipped.

```
[2]: lcs("aba", "baca")
[2]: 'aa'
```

Instead of calling functions `head` and `tail` I could have written:

```

...
elif left[0] == right[0]:
    return left[0] + lcs(left[1:], right[1:])
else:
    skip_left = lcs(left[1:], right)
    skip_right = lcs(left, right[1:])
...

```

Exercise 23.2.2 answer

```
[1]: %run -i ../m269_rec_list

def lcs_topdown(left: str, right: str) -> str:
    """Return the LCS of both strings using top-down dynamic_
    programming."""
    ...

    def lcs(left: str, right: str) -> str:
        """Auxiliary recursive function."""
        if (left, right) not in cache:
            if left == '' or right == '':
                cache[(left, right)] = ''
            elif head(left) == head(right):
                cache[(left, right)] = head(left) + lcs(tail(left),_
                tail(right))
            else:
                skip_right = lcs(left, tail(right))
                skip_left = lcs(tail(left), right)
                if len(skip_left) > len(skip_right):
                    cache[(left, right)] = skip_left
                else:
                    cache[(left, right)] = skip_right
        # print() doesn't show quote marks, so add them
        return cache[(left, right)]
    return lcs(left, right)

```

(continues on next page)

(continued from previous page)

```

        print("cache[ (", "''' + left + "''', ", ", ", "''' + right + "''',
        "''' + cache[(left, right)] + "'''")
    return cache[(left, right)]

cache = dict()
return lcs(left, right)

```

I could have written `left[0]` instead of `head(left)` and `left[1:]` instead of `tail(left)` and similarly for the right string.

Exercise 23.2.3 answer

```
[1]: def lcs_indices(left: str, right: str) -> str:
    """Return the LCS of left and right using indices, not slicing."""
    ...

    def lcs(l: int, r: int) -> str:
        """Return the LCS of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        """

        if l == len(left) or r == len(right):
            return ""
        elif left[l] == right[r]:
            return left[l] + lcs(l + 1, r + 1)
        else:
            skip_right = lcs(l, r + 1)
            skip_left = lcs(l + 1, r)
            if len(skip_left) > len(skip_right):
                return skip_left
            else:
                return skip_right

    return lcs(0, 0)
```

Exercise 23.2.4 answer

```
[1]: def lcs_topdown_matrix(left: str, right: str) -> str:
    """Return the LCS of both strings using top-down dynamic_
    programming."""

    def lcs(l: int, r: int) -> str:
        """Return the LCS of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        (continues on next page)
```

(continued from previous page)

```
"""
if cache[l][r] == None:
    if l == len(left) or r == len(right):
        cache[l][r] = ""
    elif left[l] == right[r]:
        cache[l][r] = left[l] + lcs(l + 1, r + 1)
    else:
        skip_left = lcs(l + 1, r)
        skip_right = lcs(l, r + 1)
        if len(skip_left) > len(skip_right):
            cache[l][r] = skip_left
        else:
            cache[l][r] = skip_right
print("cache[", l, "] =", "!" + cache[l][r] + "!")
)
return cache[l][r]

cache = []
for row in range(len(left) + 1): # noqa: B007
    cache.append([None] * (len(right) + 1))
return lcs(0, 0)
```

30.18.2 Knapsack

Exercise 23.3.1 answer

```
[1]: def knapsack_indices(items: list, capacity: int) -> list: # noqa:-
    ↪D103
        # docstring not repeated

    def knapsack(index: int, capacity: int) -> list:
        """Return a subsequence of items[index:].

        Preconditions: 0 ≤ index ≤ len(items) and 0 ≤ capacity
        Postconditions: the output fits the capacity and maximises
        ↪the value
        """
        if index == len(items) or capacity == 0:
            return []
        else:
            item = items[index]
            skip = knapsack(index + 1, capacity)
            if item[WEIGHT] > capacity:
                return skip
            else:
                take = [item] + knapsack(index + 1, capacity -
```

(continued from previous page)

```

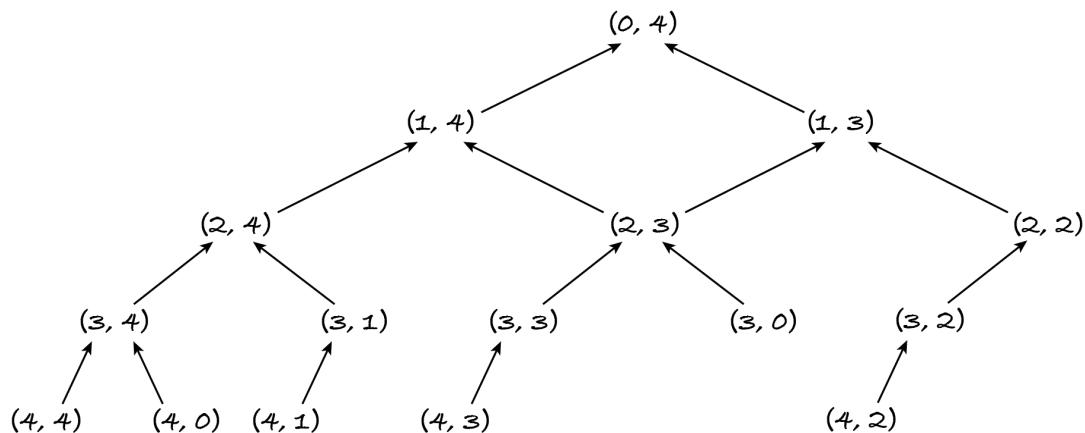
→item[WEIGHT])
        if value(skip) > value(take):
            return skip
        else:
            return take

return knapsack(0, capacity)
    
```

Exercise 23.3.2 answer

1. Here's my diagram, using indices instead of lists of items. The left child is the subproblem obtained by skipping the item at the current index. The right child is the subproblem obtained by taking the item.

Figure 23.3.1



2. Subproblems (2, 3), (3, 3), (3, 0) and (4, 3) are solved twice because each has two paths to the input instance (0, 4). Subproblem (2, 3) corresponds to skipping the first or second item and taking the other one, with weight 1, so the capacity reduces to 3. Subproblem (3, 3) is reached by skipping the third item and subproblem (3, 0) by taking it: the third item has weight 3, so the knapsack has no further capacity.

Exercise 23.3.3 answer

```

[1]: def knapsack_topdown(items: list, capacity: int) -> list: # noqa: E501
    ↪D103
    def knapsack(index: int, capacity: int) -> list:
        """Return a subsequence of items[index:].

        Preconditions: 0 ≤ index ≤ len(items) and 0 ≤ capacity
        Postconditions: the output fits the capacity and maximises
        ↪the value
        """
        if cache[index][capacity] == None:
            if index == len(items) or capacity == 0:
                cache[index][capacity] = []
            else:
                cache[index][capacity] = [
                    item
                    for item in items[index:]
                    if item[1] ≤ capacity
                ]
                for item in cache[index][capacity]:
                    cache[index][capacity].append(
                        knapsack_topdown(
                            items=item[0],
                            capacity=capacity-item[1]
                        )
                    )
            cache[index][capacity] = sorted(cache[index][capacity], key=lambda x: sum(x[1]), reverse=True)
    
```

(continues on next page)

(continued from previous page)

```

        cache[index][capacity] = []
    else:
        item = items[index]
        skip = knapsack(index + 1, capacity)
        if item[WEIGHT] > capacity:
            cache[index][capacity] = skip
        else:
            take = [item] + knapsack(index + 1, capacity -  
→item[WEIGHT])
            if value(skip) > value(take):
                cache[index][capacity] = skip
            else:
                cache[index][capacity] = take
        print("cache[", index, "][", capacity, "] =",  
→cache[index][capacity])
    return cache[index][capacity]

cache = []
for index in range(len(items) + 1):
    cache.append([None] * (capacity + 1))
return knapsack(0, capacity)

```

Exercise 23.3.4 answer

```
[1]: def knapsack_bottomup(items: list, capacity: int) -> list: # noqa:  
→D103
    cache = []
    for index in range(len(items) + 1):
        cache.append([None] * (capacity + 1))

    for index in range(len(items), -1, -1):
        for c in range(capacity + 1):
            if index == len(items) or c == 0:
                cache[index][c] = []
            else:
                item = items[index]
                skip = cache[index + 1][c]
                if item[WEIGHT] > c:
                    cache[index][c] = skip
                else:
                    take = [item] + cache[index+1][c - item[WEIGHT]]
                    if value(skip) > value(take):
                        cache[index][c] = skip
                    else:
                        cache[index][c] = take
    return cache
```

(continues on next page)

(continued from previous page)

```
return cache[0][capacity]
```

30.19 Practice 3

30.19.1 Safe places

Exercise 24.1.1 answer

We need to find shortest paths from all nodes to the assembly node.

Exercise 24.1.2 answer

... that have a shortest path to *assembly* of length $\leq \text{time}$

Exercise 24.1.3 answer

The weights aren't negative so we can apply *Dijkstra's algorithm* to find shortest paths in a weighted digraph.

Exercise 24.1.4 answer

Initialise the *safe* counter to zero. Iterate over each node of the graph. If the node isn't *assembly*, use Dijkstra's algorithm to find the shortest path (and its length) from that node to *assembly*. If the length of the path, i.e. the node's distance from *assembly*, doesn't exceed *time*, then increment *safe*.

Exercise 24.1.5 answer

The exhaustive search finds the shortest path of each of the $n - 1$ nodes to the *assembly* node, so the complexity is $O((n - 1) \times e \log e)$.

Exercise 24.1.6 answer

Create the *reverse graph* and apply Dijkstra's algorithm once to it, with *assembly* being the start node. This finds the shortest paths from *assembly* to each other node, which are the same paths as in the original graph from each node to *assembly*, but with the order of the nodes reversed.

Reversing the paths doesn't matter, since we only want to know their length, so after applying Dijkstra's algorithm on the reverse graph, iterate over all nodes to count those with a distance to *assembly* that doesn't exceed *time*.

Exercise 24.1.7 answer

My algorithm takes $\Theta(n + e)$ to reverse the graph, $O(e \log e)$ to find all shortest paths and $\Theta(n)$ to count the nodes within the required distance. The complexity is $\Theta(n + e) + O(e \log e)$.

It's possible to simplify the complexity expression to $O(n + e + e \log e) = O(n + e \log e)$ but it's more precise to indicate the complexity with a mix of Big-Theta and Big-Oh.



Info: These exercises are inspired by problem [Mice and Maze](#) from the 2001 ACM ICPC Southwestern European Regional Contest.

30.19.2 Extra staff

Exercise 24.2.1 answer

1. Computing a topological sort, to do the tasks in the correct order.
2. Kahn's algorithm.

Exercise 24.2.2 answer

For this problem, we must visit *all* nodes currently with degree zero in the same iteration, which corresponds to all possible tasks in the current week, and from them compute the nodes to visit in the next iteration, i.e. the tasks for the following week.

We therefore need two collections of nodes, for the current and for the next week's tasks. The outline of the algorithm is as follows.

1. Initialise the counter of the weeks needing additional people to zero.
2. Create a collection with the current week's tasks, i.e. with all nodes with in-degree zero, as in Kahn's algorithm.
3. While there are current tasks:
 1. If there are more current tasks than the number of people P, increment the counter.
 2. Initialise the next week's tasks with the empty collection.
 3. For each current task, decrement the degree of each of its out-neighbours and if the degree reaches zero, add that out-neighbour to the next week's tasks.
 4. Let the current tasks be the next week's tasks.
4. Finally, return the value of the counter.

Contrary to the [topological sort algorithm](#), this one doesn't remove tasks from the current week (the nodes to visit), so the collection doesn't have to be a set, queue or stack: it can be a simple sequence.

Exercise 24.2.3 answer

I've changed the variable names to better reflect the problem, but you didn't have to do it. Apart from name changes, I added three lines, indented four lines and changed another four.

```
[1]: %run -i ../m269_digraph
```

```
def extra_staff(project: DiGraph, people: int) -> int:
```

(continues on next page)

(continued from previous page)

```
"""Return how many weeks need more than people on the project.
```

Preconditions: project is acyclic, people ≥ 0

```
"""
```

```
# weeks needing extra staff
weeks = 0 # changed
```

```
# compute the initial in-degrees
```

```
indegree = dict()
for task in project.nodes():
    indegree[task] = 0
```

```
for edge in project.edges():
    indegree[edge[1]] = indegree[edge[1]] + 1
```

```
# compute the tasks that can be executed first
```

```
current_week = []
```

```
for task in project.nodes():
    if indegree[task] == 0:
```

```
    current_week.append(task)
```

```
while len(current_week) > 0:
```

```
    if len(current_week) > people: # changed
```

```
        weeks = weeks + 1 # changed
```

```
    next_week = [] # added
```

```
    for task in current_week: # added
```

```
        # simulate the removal of the executed task
```

```
        for next_task in project.neighbours(task):
```

```
            indegree[next_task] = indegree[next_task] - 1
```

```
            if indegree[next_task] == 0:
```

```
                next_week.append(next_task)
```

```
    current_week = next_week # added
```

```
return weeks # changed
```



Info: These exercises are based on problem *Hard Weeks* from the 2014 Portuguese Inter-University Programming Marathon.

30.19.3 Borrow a book

Exercise 24.3.1 answer

Finding the fewest days the book takes to get from the borrower to the lender sounds like a shortest path problem. This means we need to model the problem as a state transition graph that

represents the states the book goes through from the lender to the borrower, similar to the *rook's moves* problem. Here, it's the book that moves around.

Exercise 24.3.2 answer

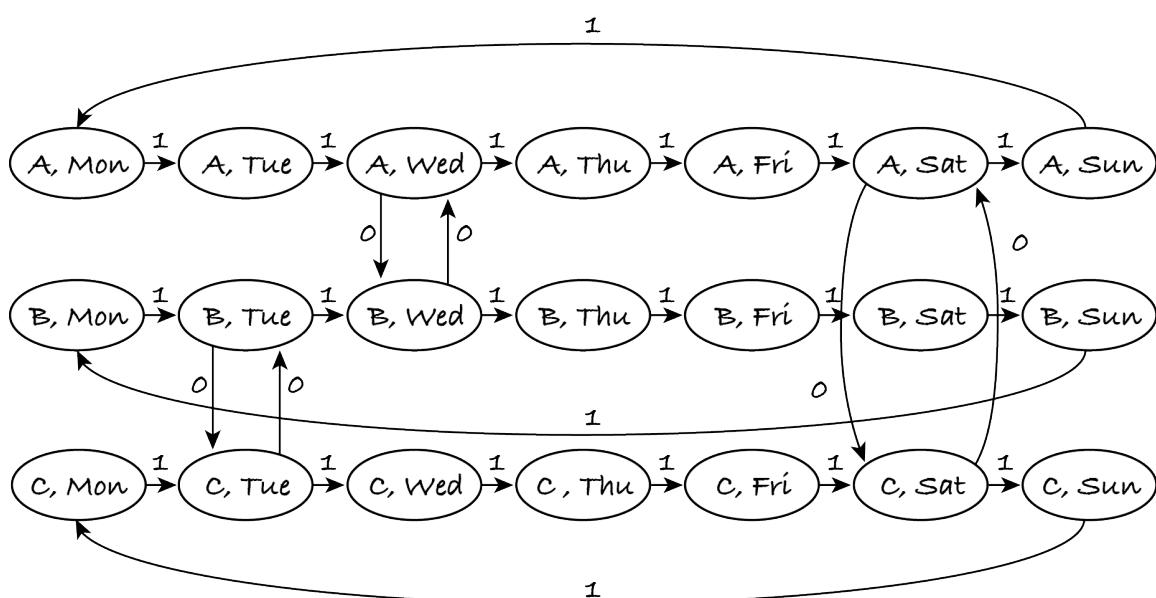
The state transition graph is a directed weighted graph. Each node is one possible state of the book. The directed edges point to the possible next states. The weight of each edge states how many days it takes for the book to go from one state to the next.

The information we need to capture is who has the book when, so the state of the book is a person–day pair. There's a transition, i.e. a directed edge, from *(person, day)* to *(person, next day)* with weight 1 if the person keeps the book from one day to the next.

If two people are at the same place on the same day D, then the book can pass from either of them to the other one: we have edges *(person 1, D) → (person 2, D)* and *(person 2, D) → (person 1, D)* with weight zero because the book is transferred on the same day.

Here's the weighted digraph for the given schedule, with Alice, Bob and Celia abbreviated as A, B and C to make the diagram more compact:

Figure 24.3.1



Once we create the graph from the schedule, we apply Dijkstra's algorithm to find all shortest paths from node *(lender, Mon)*. We stop as soon as node *(borrower, day)* is reached, for any day of the week, and return the distance of that node. If Dijkstra's algorithm ends without reaching a node for the borrower, we return infinity.

30.19.4 Levenshtein distance

Exercise 24.4.1 answer

The base cases are when either string is empty. In such cases, the distance is the number of characters of the other string, e.g. it takes three deletions to transform 'yes' into "", and two insertions to transform "" into 'no'.

- if $left = "$: $\text{edit}(left, right) = |right|$
- if $right = "$: $\text{edit}(left, right) = |left|$

If the heads are equal, we just skip them: no edit operation is needed.

- if $\text{head}(left) = \text{head}(right)$: $\text{edit}(left, right) = \text{edit}(\text{tail}(left), \text{tail}(right))$

If the heads differ, we use one operation to transform the left into the right string, but we don't know which. We may have to

- delete the left head, e.g. to turn 'rate' into 'ate'
- insert the right head, e.g. to turn 'rate' into 'grate'
- replace the left head with the right one, e.g. to turn 'rate' into 'fate'.

In each of these cases, we do one edit operation and skip one or both heads to continue processing the tails.

- otherwise: $\text{edit}(left, right) = \text{lowest of}$
 - $1 + \text{edit}(\text{tail}(left), right)$
 - $1 + \text{edit}(left, \text{tail}(right))$
 - $1 + \text{edit}(\text{tail}(left), \text{tail}(right))$

Exercise 24.4.2 answer

```
[1]: def edit(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""
    if left == "":
        return len(right)
    elif right == "":
        return len(left)
    elif left[0] == right[0]:
        return edit(left[1:], right[1:])
    else:
        delete = 1 + edit(left[1:], right)    # delete left head
        insert = 1 + edit(left, right[1:])    # insert right head
        replace = 1 + edit(left[1:], right[1:])
        return min(delete, insert, replace)
```

Exercise 24.4.3 answer

If the rest of the left string is empty, the distance is the number of characters remaining in the right string, i.e. $\text{len}([\text{right}[r:]])$.

```
[1]: def edit_indices(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""

    def edit(l: int, r: int) -> int:
```

(continues on next page)

(continued from previous page)

```

"""Return the Levenshtein distance of left[l:] and right[r:].

Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)

"""

if l == len(left):
    return len(right) - r # len(right[r:])
elif r == len(right):
    return len(left) - l # len(left[l:])
elif left[l] == right[r]:
    return edit(l + 1, r + 1)
else:
    delete = 1 + edit(l + 1, r)
    insert = 1 + edit(l, r + 1)
    replace = 1 + edit(l + 1, r + 1)
    return min(delete, insert, replace)

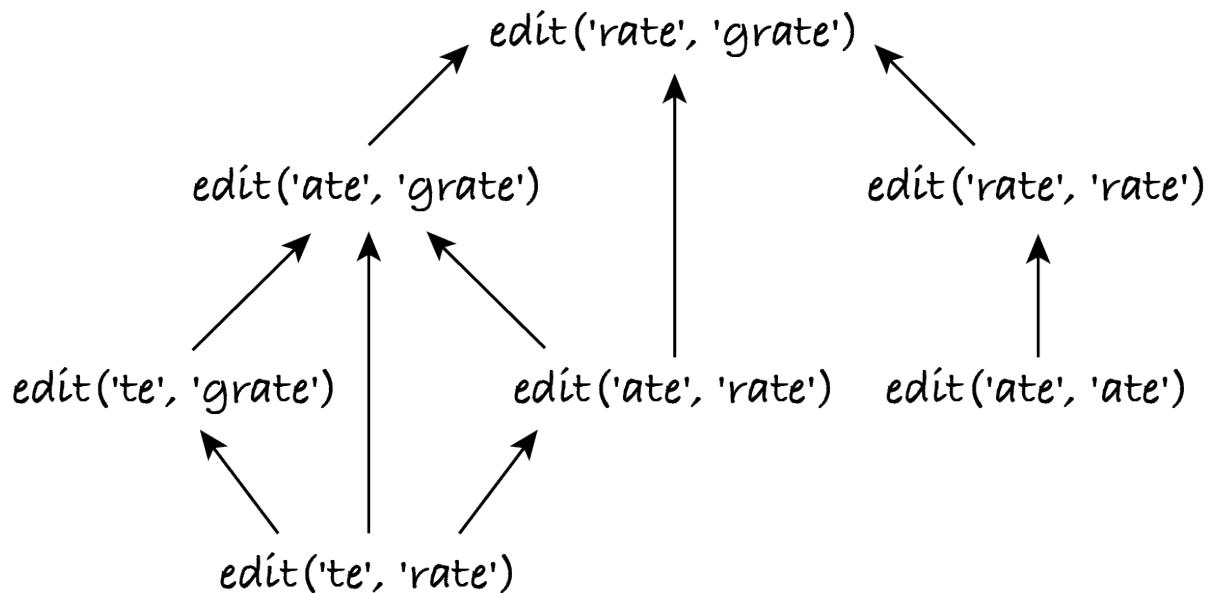
return edit(0, 0)

```

Exercise 24.4.4 answer

The following incomplete DAG shows some subproblems for one of the tests. The left and right children are obtained by skipping the left and right head, respectively, and the middle child by skipping both heads at the same time.

Figure 24.4.1



If the heads of both strings differ then the recursive algorithm skips the left one, the right one and both. Skipping first the left head and then the right head, or vice versa, is the same as skipping both at the same time, so problem instance $(tail(left), tail(right))$ or $(l+1, r+1)$ is solved several times. In general, problem instances (l, r) , $(l + 1, r)$ and $(l, r + 1)$ overlap on common subproblem $(l + 1, r + 1)$.

Exercise 24.4.5 answer

I initialise the cache with an impossible negative distance.

```
[1]: def edit_topdown(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""

    def edit(l: int, r: int) -> int:
        """Return the Levenshtein distance of left[l:] and right[r:].
        Preconditions: 0 ≤ l ≤ len(left) and 0 ≤ r ≤ len(right)
        """

        if cache[l][r] == -1:
            if l == len(left):
                cache[l][r] = len(right) - r
            elif r == len(right):
                cache[l][r] = len(left) - l
            elif left[l] == right[r]:
                cache[l][r] = edit(l + 1, r + 1)
            else:
                delete = 1 + edit(l + 1, r)
                insert = 1 + edit(l, r + 1)
                replace = 1 + edit(l + 1, r + 1)
                cache[l][r] = min(delete, insert, replace)
        return cache[l][r]

    cache = []
    for l in range(len(left) + 1):
        cache.append([-1] * (len(right) + 1))
    return edit(0, 0)
```

Exercise 24.4.6 answer

Cell `cache[l][r]` may depend on cells `cache[l][r+1]` (same line, next column), `cache[l+1][r]` (next line, same column) and `cache[l+1][r+1]` (next line, next column). When `cache[l][r]` is computed, the cells below and to the right must already be filled. We therefore fill the cache bottom-up (last to first row) and left to right within each row (last to first column).

Exercise 24.4.7 answer

```
[1]: def edit_bottomup(left: str, right: str) -> int:
    """Return the Levenshtein distance between the strings."""

    cache = []
    for l in range(len(left) + 1):
        cache.append([-1] * (len(right) + 1))

        for r in range(len(left), -1, -1):
```

(continues on next page)

(continued from previous page)

```

for r in range(len(right), -1, -1):
    if l == len(left):
        cache[l][r] = len(right) - r
    elif r == len(right):
        cache[l][r] = len(left) - 1
    elif left[l] == right[r]:
        cache[l][r] = cache[l + 1][r + 1]
    else:
        delete = cache[l + 1][r] + 1
        insert = cache[l][r + 1] + 1
        replace = cache[l + 1][r + 1] + 1
        cache[l][r] = min(replace, delete, insert)

return cache[0][0]
    
```

On my machine, this solution is about thousand times faster than the initial recursive version, with slicing.

Exercise 24.4.8 answer

Each cell is filled in constant time, with at most three cache lookups and four arithmetic operations (three additions and one call to `min`). The matrix has $(|left| + 1) \times (|right| + 1)$ cells. The worst-case complexity is $(|left| + 1) \times (|right| + 1) \times \Theta(1) = \Theta(|left| \times |right|)$.

30.19.5 Higher and higher

Exercise 24.5.1 answer

This is an optimisation problem. We must find a sequence of integers that satisfies some constraints (being a path of ascending numbers) and optimises a value: its length.

The *Trackword* problem also required finding a path in a grid. As we move in the grid, the path is incrementally extended by one square at a time, until no further squares can be visited. We can thus apply backtracking.

Exercise 24.5.2 answer

- Which of the templates applies to this problem?

This is an optimisation problem on all possible paths so I will use the template that generates permutations and keeps the best one.

- What are the candidates and the extensions?

Like for Trackword, the candidates are paths in the grid (permutations of squares) and the extensions are the sets of unvisited squares.

- When is an extension compatible with the current candidate?

If the path is empty, it can be extended with any square. Otherwise it can only be extended with a square that is vertically or horizontally adjacent and has a larger number than the last square of the path.

- What are the local and global constraints?

The next square being adjacent and having a larger number is a local constraint. There are no global constraints.

- When is a candidate a solution?

Any partial candidate can be a solution. Even if there are still extensions, none may be compatible with the candidate. If the candidate's length is the highest so far, it's the new best solution.

- What is the initial best solution?

I can't see a way to quickly compute an initial solution that's close to the best one, so I start with a path of length one: a single square. Any square will do.

Exercise 24.5.3 answer

The value being optimised is the length of the path, so I don't need to write a value function: I use the built-in `len` function.

I also don't need a function to check the global constraints: there aren't any.

```
[1]: SOLUTION = 0
VALUE = 1

def higher(grid: list) -> int:
    """Return the length of the longest path of ascending numbers in
    →grid.

    Preconditions: grid is a table of non-negative integers
                   with r > 0 rows and c > 0 columns
    """
    path = [] # initial candidate
    squares = set() # the extensions
    for row in range(len(grid)):
        for column in range(len(grid[0])):
            squares.add((row, column))
    solution = [(0, 0)] # path just with top-left square
    best = [solution, len(solution)]
    extend(path, squares, grid, best)
    return best[VALUE]

def extend(path: list, squares: set, grid: list, best: list) -> None:
    """Update best if path is a better solution, then extend it."""

```

(continues on next page)

(continued from previous page)

```

path_value = len(path)
if path_value > best[VALUE]:
    best[SOLUTION] = path
    best[VALUE] = path_value
for square in squares:
    if can_extend(square, path, grid):
        extend(path + [square], squares - {square}, grid, best)

def can_extend(square: tuple, path: list, grid: list) -> bool:
    """Check if square may extend the path towards a solution."""
    if path == []: # the empty path can be extended in any way
        return True
    last_row = path[-1][0]
    last_column = path[-1][1]
    row = square[0]
    column = square[1]
    higher = grid[row][column] > grid[last_row][last_column]
    horizontal = (row == last_row) and abs(column - last_column) == 1
    vertical = (column == last_column) and abs(row - last_row) == 1
    return higher and (horizontal or vertical)
    
```

Exercise 24.5.4 answer

Backtracking extends a candidate path but the path followed so far is irrelevant: only its last square (the square currently visited) matters. Whatever squares were visited before, the remaining path can only visit squares with a higher value than the current square.

In other words, finding the longest remaining path only depends on the current square, not those visited before. The problem becomes: what's the length of the longest ascending path from square S?

The original problem is equivalent to solving the new problem for each square in the grid and taking the largest of all lengths.

Next we have to think whether there are overlapping subproblems. There may be different paths to reach the same square S, so the longest path from S onwards will be computed several times. More precisely, if S has n neighbours with lower numbers, then there are at least n paths that reach S and problem instance 'length of longest path from S' is solved at least n times. Consider the following grid, which is one of the provided tests.

Figure 24.5.2

6	7	8
6	5	4
3	2	1

The problem instances ‘length of longest path from square with 4’ and ‘length of longest path from square with 2’ overlap on common subproblem ‘length of longest path from 5’.

To sum up, there’s only one subproblem per square but there may be several candidate paths per square because each square may be reached in several ways, which also means each subproblem may be solved several times. Since there are fewer subproblems than candidates and subproblems overlap, dynamic programming is more efficient than backtracking.

30.20 Complexity classes

30.20.1 Tractable and intractable problems

Exercise 26.1.1 answer

There’s a greedy log-linear algorithm for the interval scheduling problem, so the problem is tractable because it can be solved by a polynomial algorithm.

Exercise 26.1.2 answer

The 0/1 knapsack problem has inputs *items* (a set or sequence) and *capacity* (an integer). The output is a subset or subsequence of *items*, so it could be produced in linear time.

The dynamic programming approach has complexity $\Theta(|\text{items}|^2 \times \text{capacity})$. This is pseudo-polynomial: it’s linear in the value of *capacity* but exponential in the size of *capacity*. However, there’s no proof in this book that a better algorithm doesn’t exist.

The exact complexity is thus at least polynomial (based on the output size) and at most non-polynomial. So we can’t say if the problem is tractable or intractable.

Exercise 26.1.3 answer

In the worst case, the graph is complete and it's possible to walk directly from any place to any other place. In that case, all places except the hotel, museum and two restaurants can be visited in any order, so there are $(n - 4)!$ paths. The output size is not polynomial in the size of the input, which is $n + e$, so the problem is intractable.

Exercise 26.1.4 answer

Tractable and intractable are adjectives that apply to *problems*, not to *algorithms*. Algorithms are neither tractable nor intractable: they are polynomial or not. Pseudo-polynomial algorithms are not polynomial.

30.20.2 The P and NP classes

Exercise 26.2.1 answer

1. The certificate is a tour of the graph with total weight not exceeding w .
2. The verifier looks up the edge weights in the graph to calculate the length of the tour given by the certificate and checks that the length is at most w . This confirms that the answer is 'yes'.
3. If the graph has n nodes, then the tour has n edges. It takes $\Theta(n)$ to look up the edges' weights in the graph and add them. It takes constant time to compare the obtained length with w .

30.20.3 Reductions

Exercise 26.3.1 answer

If the sequence has n items, the maximum is the n -th smallest value. The only change is to set `solved_n = len(items)`.

Exercise 26.3.2 answer

	Unsolved	Solved
Problem:	maximum	sorting
Input:	<i>items</i> , a sequence of objects	<i>unsorted</i> , a sequence of objects
Output:	<i>largest</i> , an object	<i>sorted</i> , a sequence of objects

Input transformation (*Unsolved* to *Solved*): Let *unsorted* be *items*.

Output transformation (*Solved* to *Unsolved*): Let *largest* be the last element of *sorted*.

Both transformations take constant time and solving the sorting problem takes log-linear time. Overall, the reduction takes log-linear time.

30.21 Computability

30.21.1 Turing machine

Exercise 27.1.1 answer

Besides the ‘start’ state, we need three states ‘letter’, ‘digit’ and ‘both’ to remember what symbols have been read. When the first symbol is read, the state changes from ‘start’ to ‘letter’ or ‘digit’. As further symbols are read, the state remains the same or changes to ‘both’ when the other symbol is read.

When the current symbol is blank, the machine writes true if the state is ‘both’, otherwise it writes false. The head stays over the Boolean symbol to indicate that’s the output. The machine stays in the current state. There’s no transition for when the current symbol is a Boolean, so the machine stops.

Exercise 27.1.2 answer

```
[1]: %run -i ../m269_tm

VALID_IN = {"0", "a"}
VALID_OUT = {True, False}

is_valid = {
    ('start', 'a'):      ('a', RIGHT, 'letter'),
    ('start', '0'):      ('0', RIGHT, 'digit'),
    ('start', None):     (False, STAY, 'start'),

    ('letter', 'a'):     ('a', RIGHT, 'letter'),
    ('letter', '0'):     ('0', RIGHT, 'both'),
    ('letter', None):    (False, STAY, 'letter'),

    ('digit', 'a'):      ('a', RIGHT, 'both'),
    ('digit', '0'):      ('0', RIGHT, 'digit'),
    ('digit', None):     (False, STAY, 'digit'),

    ('both', 'a'):       ('a', RIGHT, 'both'),
    ('both', '0'):       ('0', RIGHT, 'both'),
    ('both', None):      (True, STAY, 'both'),
}

check_tm(is_valid, VALID_IN, VALID_OUT)
OK: the transition table passed the automatic checks
```

When the output is written, the machine could change to a fifth state, e.g. named ‘end’ or ‘stop’, that has no transitions. However, there’s no need for it: what makes the machine stop is the absence of transitions for Boolean symbols.

Exercise 27.1.3 answer

Each additional letter needs four new entries, one per state. The entries would be exactly like for 'a', but using 'b' instead.

To cover all 52 lowercase and uppercase letters and 10 digits, the table would have $4 \times (51 + 9) = 240$ more entries!

Exercise 27.1.4 answer

We need eight states to remember all possible combinations of conditions that have been satisfied so far:

- 'start' (no letter, digit or mark)
- 'letter', 'digit' and 'mark' (only one kind of character)
- 'letter+digit', 'letter+mark', 'digit+mark' (only two kinds of character)
- 'all' (all three kinds of character).

Apart from the initial state, the states may have any name you wish.

30.21.2 The Church–Turing thesis

Exercise 27.2.1 answer

Since 'X' may occur in the input, we need a new marker symbol that is neither a letter nor 0 nor 1, e.g. '!'. To mark and skip letters, we need one transition per letter for each state:

```
('start', 'a') : ('!', RIGHT, 'up'),  
...  
('start', 'z') : ('!', RIGHT, 'up'),  
('up', 'a') : ('a', RIGHT, 'up'),  
...  
('up', 'z') : ('z', RIGHT, 'up'),  
('back', 'a') : ('a', LEFT, 'back'),  
...  
('back', 'z') : ('z', LEFT, 'back'),
```

In addition, we must change the transition that restarts the counting, after the head returns to the last marked position. With a single marker symbol we can't remember which letter was at that position, so we simply keep the marker. The transition

```
('back', 'X') : ('a', RIGHT, 'start')
```

is changed to

```
('back', '!') : ('!', RIGHT, 'start')
```

Exercise 27.2.2 answer

We need one marker symbol per letter, so that we can replace each marker with the original letter. The markers can be negative numbers (like -1 for 'a' to -52 for 'Z'), strings (e.g. 'was a', 'was b', ...), the Unicode code of each letter (`ord('a')`, ...), and so on. The transitions related to letters become, for example:

```
('start', 'a') :      ('was a', RIGHT, 'up'), # mark and increment
→counter
...
('start', 'Z') :      ('was Z', RIGHT, 'up'),
('up', 'a') :         ('a', RIGHT, 'up'),      # skip letters
...
('up', 'Z') :         ('Z', RIGHT, 'up'),
('back', 'a') :       ('a', LEFT, 'back'),     # return to marked
→position
...
('back', 'Z') :       ('Z', LEFT, 'back'),
('back', 'was a') :   ('a', RIGHT, 'start'),   # restore letter and
→restart
...
('back', 'was Z') :   ('Z', RIGHT, 'start'),
```

30.21.3 Undecidability

Exercise 27.4.1 answer

Since SAT reduces in polynomial time to the halting problem, which in turn reduces in polynomial time to the totality problem, we have by transitivity that SAT reduces in polynomial time to the totality problem, which is therefore NP-hard too.

The same reasoning applies to the equivalence problem. Both problems are therefore NP-hard.

Exercise 27.4.2 answer

The rule

if a non-computable problem reduces to B, then B is non-computable too

cannot be applied in this case because the halting problem does *not* reduce to SAT: it's the other way round.

Alternatively, note that SAT reducing to the halting problem only means that any algorithm for the halting problem can be used to solve SAT. The absence of such an algorithm doesn't mean SAT can't be solved: it can, but it has to be by other means, not involving a reduction, namely a brute-force search over all possible interpretations.

CHAPTER 31

FIGURES

This chapter contains the descriptions of all figures throughout the book.

31.1 Sequences and iteration

31.1.1 Reversal

Figure 4.7.1

The left side of the figure shows the original list [5, True] and my left index finger pointing at 5. The right side of the figure shows my right index finger pointing at the reverse list, currently empty.

Figure 4.7.2

The left side of the figure shows the original list [5, True] with my left index finger now pointing at True. On the right side of the figure, the reverse list is now [5] and my right index finger points at 5.

Figure 4.7.3

This figure shows on the left side the original list [5, True], with my left index finger still pointing at True. On the right side, the reverse list is now [True, 5] and my right index finger continues pointing at the first item, now True.

31.2 Implementing sequences

31.2.1 Static arrays

Figure 6.2.1

The figure shows three rectangular boxes next to each other. The left-most box has numbers 123 and 124 over it and the letter H inside it. The middle box has numbers 125 and 126 over it and

the letter i inside it. The right-most box has numbers 127 and 128 over it and an exclamation mark inside it.

Figure 6.2.2

The figure shows four rows of boxes. The first row has three boxes containing numbers 123, 100542 and 1423208. These are the first addresses of the other rows of boxes. Each first row box occupies 4 bytes, indicated by the addresses above the boxes going from 1050 to 1053 for the left-hand box, 1054 to 1057 for the middle box and 1058 to 1061 for the right-hand box. The second row of boxes are the same as in the previous figure: three boxes with uppercase H, lowercase i and an exclamation mark inside them, and addresses 123 to 128 over them. The third row has two boxes, with uppercase G and lowercase o inside them, and addresses 100542 to 100545 above them. The fourth row has four boxes, with lowercase l, e, f and t inside, and addresses 1423208 to 1423215 above them.

Figure 6.2.3

The figure shows four separate groups of boxes. Three of the groups contain the words Hi!, Go and left, with one character per box. The fourth group has three boxes, each containing the start of an arrow. Each of the three arrows ends below the first box of one of the other three groups. The arrow starting in the left-hand box points to H (the first box of Hi!), the arrow starting in the middle box points to G (the first box of Go) and the arrow starting in the right-hand box points to l (the first box of left).

Figure 6.2.4

The figure shows variable names pair, point, x and y. Names pair and point have arrows next to them leading to a pair of boxes. The first box has an arrow that leads to a separate box, with number 1 in it. The second box has an arrow that leads to a separate box, with number 2 in it. Variable name x has an arrow that leads to the box with the number 1. Variable y has an arrow that leads to the box with the number 2.

Figure 6.2.5

This figure is the same as the previous one, but the arrow next to variable y is no longer pointing at the box with number 2. Instead, it's pointing at a new box with number 3.

Figure 6.2.6

This figure is the same as the previous one, but the arrow stemming from the second of the pair of boxes is no longer pointing at the box with number 2. Instead, it's pointing at yet another new box with number 3, so there are two boxes with number 3: one has an arrow from variable y, the other has an arrow from the second of the pair of boxes. The box with number 2 has no arrows leading to it.

31.2.2 Bounded sequences

Figure 6.4.1

This figure shows a variable named ‘items’ pointing to a series of five boxes, each with its index beneath, from 0 to 4. A variable named ‘size’ points to index 3. The first box points to another box with value True. The third box points to another box with value 0.2. The other three boxes all point to the same box with value None.

31.2.3 Linked lists

Figure 6.7.1

The figure shows a linked list with three nodes, each depicted by a pair of boxes. The variable ‘head’ points to (i.e. has an arrow to) a pair of boxes: the left box has number 0; the right box has an arrow leading to another pair of boxes. This second pair has number 1 in the left box and in the right box an arrow to another pair of boxes. This third pair of boxes has number 2 in the left box and a cross in the right box.

Figure 6.7.2

This figure shows two linked lists. The first list is the same as in the previous figure, with three nodes with numbers 0, 1 and 2 and with variable ‘head’ pointing to the first node with number 0. There are two more variables: ‘before’ points to the second node, which has number 1, and variable ‘after’ points to the third node, which has number 2. The second list has a single node pointed to by variable ‘new’. The node has number 3 in the left box and a cross in the right box.

Figure 6.7.3

The figure is the same as the previous one, but with two differences. First, the arrow going from the node with number 1 to the node with number 2 now goes to the node with number 3. Second, the cross in the node with number 3 has been replaced with an arrow going to the node with number 2.

Figure 6.7.4

The figure shows two linked lists, each with one node. One node has the number 0 in the left box and a cross in the right box. The other node has an exclamation mark in the left box and a cross in the right box. The variable ‘new’ points to the second node, with the exclamation mark. The variables ‘head’ and ‘before’ both point to the first node, with number 0. Additionally, the variable ‘after’ points to a separate x, indicating that ‘after’ is a null pointer.

Figure 6.7.5

This figure is like the previous one except that the cross in the right half of the node with number 0 has been replaced with an arrow that leads to the other node, with the exclamation mark. The two nodes now form part of a single linked list.

31.3 Stacks and queues

31.3.1 Stacks

Figure 7.1.1

The left half of the figure shows a linked list representation of a stack. There's an arrow from the word 'length' to a box with number 3 inside. There's an arrow from the word 'head' to a pair of boxes: the left box has number 2, the right box has an arrow leading to another pair of boxes. This pair has number 1 in the left box and another arrow to another pair of boxes. This third pair of boxes has number 0 in the left box and a cross in the right box. The right half of the figure shows an array representation of the same stack. The array is depicted as a sequence of 5 boxes with their indices, from 0 to 4, written beneath. The first three boxes have numbers 0, 1 and 2 inside them, and the last two boxes have the word 'None'. There's an arrow from the word 'items' to the first box, with number 0. There's an arrow from word 'length' to index 3.

31.3.2 Queues

Figure 7.3.1

The left half of the figure shows the word Alice surround by the numbers 1, 2, 4, 5 and 6, going clockwise, with 1 at the 12 o'clock position. There's an arrow from Alice to number 4. The right half of the figure shows a series of boxes, with the numbers 4, 5, 6, 1, 2 inside them, from left to right. There's an arrow going from the word 'front' to the left-most box, with number 4.

31.4 Unordered collections

31.4.1 Hash tables

Figure 8.3.1

The figure shows three boxes, with indices 0, 1 and 2 next to them, and with numbers 312, 407 and 566 inside, respectively. There are three arrows leading respectively from name Alice to index 0, from name Bob to index 1 and from name Carol to index 3. Above the three arrows there's the word 'hash'.

Figure 8.3.2

The figure shows three boxes, with indices 0, 1 and 2 next to them, three names (Alice, Bob and Carol) with arrows to the indices, and the word 'hash' above the arrows. Each of the boxes has a further arrow, each leading to a separate box. Carol has an arrow to the box with index 0, which in turn has an arrow to a box with the tuple (Carol, 566) in it. Alice and Bob both have arrows leading to the box with index 1, which in turn has an arrow to a box divided in two: the left half has the tuple (Bob, 407) and the right half has the tuple (Alice, 312). The box with index 2 has no arrow leading from a name to it, but it has an arrow to an empty box.

Figure 8.3.3

The figure shows six boxes, with indices 0 to 5 next to them, three names (Alice, Bob and Carol) with arrows to some indices, and the word ‘hash’ above the arrows. Each of the boxes has a further arrow, each leading to a separate box. Alice has an arrow to the box with index 4, which in turn has an arrow to a box with the tuple (Alice, 312) in it. Bob has an arrow to the box with index 1, which in turn has an arrow to a box with the tuple (Bob, 407) in it. Carol has an arrow to the box with index 3, which in turn has an arrow to a box with the tuple (Carol, 566) in it. The boxes with indices 0, 2 and 5 have no arrows leading from a name to them, but each one has an arrow to a different empty box.

31.5 Practice 1

31.5.1 Trains

Figure 9.4.1

The figure shows a vertical rail track that bifurcates into a left and a right track, both horizontal and both with a left-pointing arrow beneath them. The right track has three wagons on it, labelled 1, 2 and 3 from left to right. The left track has three wagons labelled 1, 3, 2 from left to right. At the bottom of the vertical track is the word Station.

31.6 Exhaustive search

31.6.1 Searching permutations

Figure 11.4.1

This figure shows four circles, labelled 0 to 3, and a straight line between each pair of circles. Each line is labelled with an integer denoting the cost of travelling between those places. It costs 15 to travel between 0 and 1, 10 between 0 and 2 and between 2 and 3, 5 between 0 and 3 and between 3 and 1, and 30 between 1 and 2.

Figure 11.4.2

This figure shows a line chart with n going from 1 to 7 in the x-axis. The y-axis goes from 0 to 5000. There are three lines in the chart. The black line plots the values for the square of n (n to the power of 2). The blue line plots the values for the cube of n (n to the power of 3). The red line plots the values for the factorial of n . The black line is horizontal. The blue line goes above the black line from $n = 3$ onwards and keeps rising gently. The red line is below the blue line until $n = 5$. Then it rises abruptly. The factorial of 6 is almost 1000 and the factorial of 7 is over 5000, while the cube of 7 is clearly under 500.

31.7 Sorting

31.7.1 Merge sort

Figure 14.5.1

This diagram shows seven rows of sequences of letters. When a sequence is split, it is connected with lines to the two resulting sequences in the next row. When two sequences are merged, they are connected with straight lines to the resulting sequence in the next row. The first row shows sequence SORTING. The second row shows it has been split into SORT and ING. The third row shows the first sequence is split into SO and RT, and the second into IN and G. In the fourth row, SO, RT and IN are further split into single letters. In the fifth row, S and O are merged into OS, R and T into RT, and I and N into IN. In the fifth row, OS and RT are merged into ORST, and IN is merged with G into GIN. In the final seventh row, ORST and GIN are merged into GINORST, the sorted sequence of letters.

31.7.2 Quicksort

Figure 14.6.1

This diagram shows seven rows of sequences of letters. When a sequence is split, it is connected with lines to the two resulting unsorted sequences and the pivot in the next row. When two sorted sequences and the pivot are concatenated, they are connected with straight lines to the resulting sequence in the next row. The first row shows sequence SORTING. The second row shows it has been split into ORING and T, with pivot S. The third row shows that ORING is split into ING and R, with pivot O. The fourth row shows that ING is split into G and N with pivot I. These are concatenated into GIN. The sixth row concatenates GIN and R around pivot O to obtain GINOR. The seventh row concatenates GINOR and T around pivot S to obtain GINORST.

31.8 Rooted trees

31.8.1 Binary tree

Figure 16.1.1

This figure shows a diagram made of circles connected by lines. Each circle surrounds an arithmetic operator or a number. Each circle with an operator is connected by two lines to two other circles below it, to the left and right. Each circle with a number has no further circles below it. The circle at the top has the subtraction operator. Below it, the left circle has the multiplication operator and the right circle has number 6. Below the multiplication operator, the left circle has the addition operator and the right circle has number 5. Below the addition operator, the left circle has number 3 and the right circle has number 4.

Figure 16.1.2

This figure shows three separate circle-and-line diagrams, with the same operators and numbers as the previous figure. Again, only circles with operators are connected to a left and a right circle below. Left diagram: The top circle has the multiplication operator. The circles below have the addition operator on the left and the subtraction operator on the right. Below the addition circle

are the numbers 3 on the left and 4 on the right. Below the subtraction circle are the numbers 5 on the left and 6 on the right. Middle diagram: The top circle has the addition operator. The circles below have number 3 on the left and the subtraction operator on the right. Below the subtraction circle are the multiplication operator on the left and number 6 on the right. Below the multiplication circle are the numbers 4 on the left and 5 on the right. Right diagram: The top circle has the subtraction operator. The circles below have the addition operator on the left and number 6 on the right. Below the addition circle are number 3 on the left and the multiplication operator on the right. Below the multiplication circle are the numbers 4 on the left and 5 on the right.

31.8.2 Traversals

Figure 16.3.1

This figure shows a diagram made of circles connected by lines. Each circle surrounds an arithmetic operator or a number. Each circle with an operator is connected by two lines to two other circles below it, to the left and right. Each circle with a number has no further circles below it. The circle at the top has the subtraction operator. Below it, the left circle has the multiplication operator and the right circle has number 6. Below the multiplication operator, the left circle has the addition operator and the right circle has number 5. Below the addition operator, the left circle has number 3 and the right circle has number 4.

31.8.3 Binary search trees

Figure 16.4.1

This figure shows two separate circle-and-line diagrams. Each circle has a number. Left diagram: The top circle has number 4. Below this are two circles. The circle on the left contains number 1; the circle on the right contains number 8. The circle with 8 has a single circle below, to its left, with number 5, which in turn also has a single circle below, to its right, with number 6. Right diagram: This is the same diagram as on the left, except that numbers 5 and 6 have swapped the circles they're in.

Figure 16.4.2

This figure shows in the middle a right-pointing horizontal arrow, with the word ‘becomes’ above it. To the left of the arrow there’s a circle enclosing the word ‘root’. There’s nothing to the right of the arrow.

Figure 16.4.3

This figure shows in the middle a right-pointing horizontal arrow, with the word ‘becomes’ above it. To the left of the arrow is a circle enclosing the word ‘root’. Below the circle, to its right, is a triangle with the letter R inside it. The top tip of the triangle is attached with a line to the circle. To the right of the arrow is a triangle with the letter R inside it.

Figure 16.4.4

This figure shows in the middle a right-pointing horizontal arrow, with the word ‘becomes’ above it. To the left of the arrow is a circle enclosing the word ‘root’. The circle has two lines connecting to two triangles below it. The left triangle has the letters L and p inside it: the p is in the bottom right corner of the triangle. The right triangle has the letters R and s inside it: the s is in the bottom left corner of the triangle. To the right of the arrow is the same diagram as on the left, except that the word ‘root’ has been replaced by the letter s inside the circle, and the right triangle only has the letter R. The triangle’s left corner has been chopped off.

31.8.4 Balanced trees

Figure 16.5.1

This circle-and-line diagram has empty circles. Each circle has two circles below it, to the left and right, except the circles in the fourth row. There are one circle in the top row, two circles in the second row, four circles in the third row, and eight circles in the fourth row.

Figure 16.5.2

This figure shows two separate circle-and-line diagrams. Each circle has a number next to it, outside the circle. Left diagram: The top circle has number 5 inside and number 0 outside. The left circle below has number 4 inside and number 1 outside; the right circle has number 8 inside and number 1 outside. Both circles have a single circle below, to their left. The circle below 4 has number 1 inside and number 0 outside; the circle below 8 has number 6 inside and number 0 outside. Right diagram: The top circle has the addition operator inside and number -2 outside. The left circle below has number 3 inside and number 0 outside. The right circle below addition has the subtraction operator inside and number 1 outside. The left circle below subtraction has the multiplication operator inside and the number 0 outside. The right circle below subtraction has number 6 inside and number 0 outside. The left and right circles below multiplication have numbers 4 and 5 inside and both have number 0 outside.

31.8.5 Heapsort

Figure 16.6.1

This figure shows two separate binary trees. The nodes are empty: there’s nothing inside the circles. In both trees, the root has two children. In the left-hand tree, the left child of the root has two children but the right child of the root has no children. In the right-hand tree, the left child of the root has one left child and the right child of the root has one right child.

Figure 16.6.2

This figure shows three binary trees. Left tree: The root, with key 0, has two children with keys 1 and 2. The node with key 1 has two children with keys 3 and 4. Middle tree: The root, with key 0, has two children with keys 3 and 2. The node with key 3 has two children with keys 1 and 4. Right tree: The root, with key 4, has two children with keys 3 and 2. The node with key 3 has two children with keys 1 and 0.

Figure 16.6.3

This figure shows three binary trees, each with three nodes: the root and two children. The middle tree's root has key 4. Its left child has key 1 and its right child has key 2. In the left-hand tree, 4 and 1 have swapped places: 1 is in the root, 4 is in the left child and 2 stays as the right child. In the right-hand tree, 4 and 2 have swapped places: 2 is in the root, 4 is in the right child and 1 stays as the left child.

31.9 Graphs 1

31.9.1 Modelling with graphs

Figure 17.1.1

The diagram on the left consists of four ellipses, labelled Alice, Bob, Chan and David, and four straight lines between them. Bob is connected with Alice, Chan and David. Alice and David are also connected. The diagram on the right consists of four circles labelled 1, 2, 3 and 4, and four arrows connecting them. There are two arrows from 4 to 1 and 2. There's one arrow from 1 to 2 and another from 2 to 1.

Figure 17.1.2

This screenshot shows the top left-hand corner of a spreadsheet, with three columns labelled A, B and C from left to right and two rows numbered 1 and 2. Column A has the text 'Cost' in row 1 and the number 5 in row 2. Column B has the text 'VAT' in row 1 and the formula '=A2 times 0.2' in row 2. Column C has the text 'Total' in row 1 and the formula '=A2 + B2' in row 2.

Figure 17.1.3

This diagram has three circles, labelled A2, B2 and C2, from left to right. There are three arrows, from A2 to B2, from B2 to C2, and from A2 to C2.

Figure 17.1.4

This diagram has four Noughts and Crosses boards, each being a three by three square grid. The board at the top of the diagram is empty. The board on the left has a single X, in the top left square. The board on the right has a single O, in the centre square. The board at the bottom has an X in the top left square and an O in the centre. Two arrows go from the top empty board to the left and right boards. Two other arrows go from the left and right boards to the bottom one.

Figure 17.1.5

This figure shows two layouts of the same graph with six nodes labelled A, B, N, S, W and E. In the left-hand layout, nodes N, S, W, E are in the north, south, west and east positions, respectively. Node A is below node N. Node B is below node A, between nodes W and E, and above node S. In the right-hand layout, node A is to the left of node B. Nodes N and S are between A and B, with N above them and S below them. Node W is above node E and both are to the right of node B. Both layouts have nine undirected edges connecting the nodes as follows.

Node N is connected to A, B and S. Node S is also connected to A and B. A is also connected to B. W and E are connected to each other and to B.

Figure 17.1.6

This diagram has three ellipses, arranged horizontally and labelled from left to right as ‘first item’, ‘second item’ and ‘last item’. There are three dots between the second and third ellipse, i.e. between the second and last items. There are three arrows going from the first to the second items, from the second item to three dots and from the three dots to the last item.

31.9.2 Basic concepts

Figure 17.2.1

The diagram on the left consists of four ellipses, labelled Alice, Bob, Chan and David, and four straight lines between them. Bob is connected with Alice, Chan and David. Alice and David are also connected. The diagram on the right consists of four circles labelled 1, 2, 3 and 4, and four arrows connecting them. There are two arrows from 4 to 1 and 2. There’s one arrow from 1 to 2 and another from 2 to 1.

Figure 17.2.2

This figure shows two circle-and-line diagrams with six circles, labelled A to F. In the left diagram, A has a line down to B, which has a line down to C. B also has a line to E on its right. E has a line up to D and another down to F. In the right diagram, E is at the top. Below it are B, D and F, from left to right, each connected to E with a line. Below B are A and C, from left to right, both connected to B with a line.

Figure 17.2.3

This figure shows several diagrams, arranged in four rows and four columns. Columns 1, 2, 3 and 4, from left to right, show graphs with one, two, three and four nodes, respectively. The first row shows null graphs: they are depicted as one, two, three or four circles, without any lines between them. The second row shows path graphs. The first diagram has a single circle. The second has two circles connected by a line. The third has three circles arranged horizontally, with a line between the first two, and another line between the second and third circles. The fourth diagram has four circles arranged horizontally, and three lines, each connecting a circle to the next. The third row shows cycle graphs. The first two columns are empty. The third column has three circles arranged horizontally, with three lines: the first circle is connected to the second, the second to the third, and the third to the first. The fourth column has four circles arranged horizontally, with four lines: each circle is connected to the next, and the last is connected to the first. The fourth row shows complete graphs. The first diagram has a single circle. The second has two circles connected by a line. The third has three circles arranged in a triangle, with three lines: each circle is connected to the other two. The fourth diagram has four circles arranged in a rectangle, with six lines: each circle is connected to the other three.

31.9.3 Edge list representation

Figure 17.3.1

The diagram on the left consists of four ellipses, labelled Alice, Bob, Chan and David, and four straight lines between them. Bob is connected with Alice, Chan and David. Alice and David are also connected. The diagram on the right consists of four circles labelled 1, 2, 3 and 4, and four arrows connecting them. There are two arrows from 4 to 1 and 2. There's one arrow from 1 to 2 and another from 2 to 1.

31.9.4 Adjacency matrix representation

Figure 17.4.1

The diagram on the left consists of four ellipses, labelled Alice, Bob, Chan and David, and four straight lines between them. Bob is connected with Alice, Chan and David. Alice and David are also connected. The diagram on the right consists of four circles labelled 1, 2, 3 and 4, and four arrows connecting them. There are two arrows from 4 to 1 and 2. There's one arrow from 1 to 2 and another from 2 to 1.

31.9.5 Adjacency list representation

Figure 17.5.1

The diagram on the left consists of four ellipses, labelled Alice, Bob, Chan and David, and four straight lines between them. Bob is connected with Alice, Chan and David. Alice and David are also connected. The diagram on the right consists of four circles labelled 1, 2, 3 and 4, and four arrows connecting them. There are two arrows from 4 to 1 and 2. There's one arrow from 1 to 2 and another from 2 to 1.

31.10 Greed

31.10.1 Interval scheduling

Figure 18.1.1

This figure has a timeline at the bottom, going from 0 to 7 from left to right. There are four time intervals, depicted as horizontal lines. From left to right the intervals are: start at 1 and end at 4; start at 3 and end at 7; start at 5 and end at 6; start at 6 and end at 7.

Figure 18.1.2

This figure shows three horizontal lines labelled A, B and C. Line C starts before A and B, and ends after them. Line A starts and ends before B.

Figure 18.1.3

This figure shows three horizontal lines labelled A, B and C. Line A starts and ends first. Line B starts end ends last. Line C fills a short gap between A and B: it starts where line A ends and ends where line B starts. Line C is the shortest; line B is longer; line A is the longest.

Figure 18.1.4

This diagram shows nine horizontal lines arranged in three rows. The first row has four lines, labelled A, B, C and D from left to right. The second row has three lines labelled E, F and G from left to right. Line E starts in the middle of line A and ends in the middle of line B. Line F starts before the end of B and ends in the middle of C. Line G starts before the end of C and ends in the middle of D. The third row has two lines labelled H and I. Line H is left-aligned with line E but ends before line B starts. Line I is left-aligned with line D and ends after line G.

Figure 18.1.5

This diagram has eight horizontal lines representing time intervals, labelled from A to H. There's a dashed vertical line labelled t. Lines A, B and C start at different points but all end at t. Lines D and E cross the dashed line: they start before t and end after t. Lines F, G and H all start during interval D but after t. Lines G and H start after E. Lines F and H end at the same time, before G.

31.10.2 Weighted graphs

Figure 18.2.1

This diagram shows a right-angled triangle, with the hypotenuse going from the left bottom point labelled (x_1, y_1) to the top right point labelled (x_2, y_2) . The horizontal side of the triangle is labelled m_x and the vertical side is labelled m_y . The hypotenuse is labelled with the equation $e = \sqrt{\text{square root of the sum of the square of } m_x \text{ and the square of } m_y}$.

Figure 18.2.2

This figure shows four circles, labelled 0 to 3, and a straight line between each pair of circles. Each line is labelled with an integer denoting the cost of travelling between those places. It costs 15 to travel between 0 and 1, 10 between 0 and 2 and between 2 and 3, 5 between 0 and 3 and between 3 and 1, and 30 between 1 and 2.

31.10.3 Minimum spanning tree

Figure 18.3.1

This figure has two circle-and-line diagrams. Both diagrams have four circles labelled A, B, C and D, laid out clockwise from A on the left. Each diagram has five lines connecting the nodes. Each line is labelled with an integer. Line AB has label 1. Line BC has label 4. Line CD has label 5. Line DA has label 2. Line BD has label 2. In the left-hand diagram, lines DA and DC, labelled 2 and 5, are dashed. In the right-hand diagram, lines DB and DC, labelled 2 and 5, are dashed.

Figure 18.3.2

This figure shows a circle-and-line diagram representing a tree. The diagram has a vertical dashed line that crosses a single edge, labelled M. Among the nodes to the left of the dashed line are two labelled start and A. Among the nodes to the right of the dashed line there's one labelled B.

Figure 18.3.3

This figure shows four circles, labelled 0 to 3, and a straight line between each pair of circles. Each line is labelled with an integer denoting the cost of travelling between those places. It costs 15 to travel between 0 and 1, 10 between 0 and 2 and between 2 and 3, 5 between 0 and 3 and between 3 and 1, and 30 between 1 and 2.

31.10.4 Shortest paths

Figure 18.4.1

This diagram has three circles labelled start, A and B. A dashed arrow, labelled ta, goes from start to A. Another dashed arrow, labelled tb, goes from start to B. A non-dashed arrow, labelled w, goes from B to A.

31.11 Practice 2

31.11.1 Beams

Figure 19.3.1

This circle-and-line diagram shows an undirected unweighted graph with six nodes, labelled from 1 to 6, and six edges. The nodes are organised in two rows. In the top row, nodes 1 and 2 are connected, and nodes 3 and 4 are connected. In the bottom row, node 5 is connected to node 2 above it, and node 6 is connected to nodes 2, 3 and 4.

Figure 19.3.2

This circle-and-line diagram shows an undirected unweighted graph with six nodes, labelled from 1 to 6, and six edges. The nodes are organised in two rows. In the top row, nodes 1 and 2 are connected, and nodes 3 and 4 are connected. In the bottom row, node 5 is connected to node 2 above it, and node 6 is connected to nodes 2, 3 and 4.

Figure 19.3.3

This circle-and-line diagram shows an undirected unweighted graph with four nodes and two edges. Nodes 1 and 2 are connected. Nodes 3 and 4 are connected.

31.11.2 Up and down

Figure 19.4.1

This figure shows three line charts plotting values in the y-axis against indices in the x-axis. Each line chart plots five values, indicated by small black circles. The first value, at index 0, is plotted on the y-axis. Consecutive circles are linked by a straight line to show the increase or decrease in value. The largest value, i.e. the highest black circle, is highlighted with a further circle around it. In the left-hand chart the values increase from left to right, i.e. each further black circle is higher up. The last one is highlighted. In the middle chart the values decrease from left to right. The first value is highlighted because it's the highest one. In the right-hand chart the values first go up and then go down. The highest value, which is neither the first nor the last, is highlighted.

Figure 19.4.2

This figure shows three line charts plotting values in the y-axis against indices in the x-axis. Each line chart plots five values, indicated by small black circles. The first value, at index 0, is plotted on the y-axis. The third and fourth values are marked M and M+1 on the x-axis. Consecutive circles are linked by a straight line to show the increase or decrease in value. The largest value, i.e. the highest black circle, is highlighted with a further circle around it. In the left-hand chart the values increase from left to right, i.e. each further black circle is higher up. The last one is highlighted. In the middle chart the values decrease from left to right. The first value is highlighted because it's the highest one. In the right-hand chart the values first go up and then go down. The highest value, which is neither the first nor the last, is highlighted.

31.11.3 A knight goes places

Figure 19.5.1

This figure shows a chessboard made of squares arranged in five rows, numbered zero to four from top to bottom, and six columns, numbered zero to five from left to right. The square in row 3, column 4 (second row from the bottom, second column from the right) has the horse-shaped image of a knight. Each other square has a number from one to four. There are L-shaped lines with arrows from the square with the knight to the four squares with number 1. Two lines go up and then left or right. The other two lines go left and then up or down.

Figure 19.5.2

This figure shows a chessboard with three rows and three columns. The knight is in the top left-hand corner. The central square is empty. Each other square has a number from 1 to 4.

31.12 Graphs 2

31.12.1 Undirected graph components

Figure 21.1.1

This figure shows an undirected graph with six nodes, labelled A to F, and three edges: between A and B, between A and C, and between D and E.

31.12.2 Directed graph components

Figure 21.2.1

This figure shows a digraph with six nodes, labelled A to F, and four edges: from A and B, from B to C, from C to A, and from D to E.

Figure 21.2.2

This figure shows an undirected graph with six nodes, labelled A to F, and four edges: between A and B, between B and C, between C and A, and between D and E.

Figure 21.2.3

This figure shows a digraph with six nodes, labelled A to F, and four edges: from C to B, from B to A, from A to C, and from E to D.

31.12.3 Topological sort

Figure 21.3.1

This figure shows a directed graph with three nodes, labelled A2, B2 and C2, from left to right. There are three directed edges, pointing left to right, from A2 to B2, from B2 to C2, and from A2 to C2.

Figure 21.3.2

This figure shows a directed graph with four nodes, labelled A to D, and two edges, from A to B and from C to D.

Figure 21.3.3

This figure shows a directed graph with three nodes, labelled A to C, from left to right. There are three edges. The edges from A to B and from B to C point to the right. The edge from C to A points to the left.

Figure 21.3.4

This figure has four separate panes, each showing a digraph and a sequence of nodes. From left to right, the panes are as follows. Pane 1: The digraph has three nodes A2, B2, C2 and three edges, from A2 to B2, from B2 to C2 and from A2 to C2. Node A2 has number zero next to it, node B2 has number 1, and node C2 has number 2. The sequence is empty. Pane 2: The digraph has no node A2. There's only one edge from B2 to C2. Node B2 has number 0 and node C2 has number 1. The sequence is just A2. Pane 3: The digraph no longer has B2. There's only node C2, with number 0, and no edges. The sequence is A2, B2. Pane 4: There's no digraph. The sequence is A2, B2, C2.

Figure 21.3.5

This figure has four separate panes, each showing a digraph and a sequence of nodes. The digraph is always the same: it has three nodes, labelled A2, B2 and C2, and three edges, from A2 to B2, from B2 to C2 and from A2 to C2. From left to right, the panes are as follows. Pane 1: Node A2 has number zero next to it, node B2 has number 1, and node C2 has number 2. The sequence is empty. Pane 2: Nodes A2 and B2 have number zero next to them and node C2 has number 1. The sequence is just A2. Pane 3: All nodes have number 0 next to them. The sequence is A2, B2. Pane 4: All nodes have number 0 next to them. The sequence is A2, B2, C2.

31.12.4 State graphs

Figure 21.4.1

This figure shows a chess board of four rows of four squares each. Both the rows and columns are numbered from 0 to 3. The top left square, with coordinates (0, 0), has a tower icon, representing a rook. There are four arrows drawn on the board. The first goes right from square (0, 0) to square (0, 1). The second goes right from (0, 1) to (0, 3). The third goes down from (0, 3) to (3, 3). The fourth goes left from (3, 3) to square (2, 3), which has the letter E.

Figure 21.4.2

This figure shows three undirected graphs, all with the same 16 nodes, labelled (0, 0) to (3, 3) and laid out in a four by four grid, like the squares they correspond to. For example, the four corner nodes are (0, 0) in the top left, (0, 3) in the top right, (3, 0) in the bottom left and (3, 3) in the bottom right. Left-hand graph: Every node is connected to the two, three or four nodes horizontally and vertically around it. For example, node (0, 0) has two edges: to node (0, 1) on the right and node (1, 0) below. Node (1, 1) has four edges: to node (0, 1) above, node (2, 1) below, node (1, 0) on the left and node (1, 2) on the right. Middle graph: Each node is connected to exactly two nodes: one is in the same row but two columns to the left or right and the other is in the same column but two rows above or below. For example, node (0, 1) is connected to (0, 3), which is two columns to the right, and to (2, 1), which is two rows below. Right-hand graph: This graph only has eight edges. There are four vertical edges, each connecting a node in the top row and the node in the same column of the bottom row, e.g. (0, 2) and (3, 2). There are four horizontal edges, between each node of the first column and the node in the same row of the third column, e.g. (1, 0) and (1, 3).

Figure 21.4.3

This figure shows a directed graph with 12 nodes, arranged in three rows of four columns. Each node is labelled with one of the four squares (0, 0) to (0, 3), from the first to the last column, and one of the distances 1 to 3, from the top to the bottom row. For example, the first row nodes are ((0, 0), 1), ((0, 1), 1), ((0, 2), 1) and ((0, 3), 1) and the left column nodes are ((0, 0), 1), ((0, 0), 2) and ((0, 0), 3). There are 12 directed edges in total. There are two cycles of three nodes each. One goes from ((0, 0), 1) to ((0, 1), 2) then ((0, 3), 3) and back to ((0, 0), 1). The other goes from ((0, 3), 1) to ((0, 2), 2) then ((0, 0), 3) and back to ((0, 3), 1). Another four edges are part of two paths of three nodes each. One goes from ((0, 1), 1) to ((0, 0), 2) and then ((0, 2), 3). The other goes from ((0, 2), 1) to ((0, 3), 2) and then ((0, 1), 3). In addition there's an edge from ((0, 1), 1) to ((0, 2), 2) and an edge from ((0, 2), 1) to ((0, 1), 2).

31.13 Backtracking

31.13.1 Generate sequences

Figure 22.1.1

This figure shows a tree in which each node has a sequence and a set. Each node has one child for each element in the set: the child node has that element at the end of the sequence and no longer in the set. The root node has the empty sequence and set {1, 2, 3}. Each further level of the tree increases the size of the sequence and decreases the size of the set. Level 1, the root, has zero numbers in the sequence and three numbers in the set. Level 2 nodes have one number in the sequence and two in the set. Level 3 nodes have two numbers in the sequence and one in the set. Level 4 nodes, the leaves, have three numbers in the sequence and none in the set. For example, the root's three children are: sequence (1) and set {2,3}; sequence (2) and set {1,3}; sequence (3) and set {1,2}. Node (1) {2,3} has two children: node (1,2) {3} which in turn has a single child (1,2,3) {}, and node (1,3) {2} which in turn has a single child (1,3,2) {}.

31.13.2 Trackword

Figure 22.3.1

This figure shows four grids of 3 by 3 squares with letters. Grid 1: The letters are S, E and T in the first row, E, R and R in the second row, and V, E and I in the third row. Grid 2: This grid has the same letters and a line showing the path that spells out the word 'retrieves'. The line starts at R in the central square, goes up to E, then goes down the right-hand column (T, R and I), then goes left along the bottom row (E and V) and finally up the left-hand column (E and S). Grid 3: This grid has a line showing one path that spells out 'tree'. The line starts at the T in the top right corner, goes diagonally to R in the central square, then left to E, then diagonally up right to E. Grid 4: This grid shows another path that spells out 'tree'. The line starts again at the T in the top right corner, but then moves down to R, diagonally up left to the E in the first row and finally diagonally down left to the E in the left-hand column. This and the previous path use the same two E squares, but in a different order.

31.13.3 Back to the TSP

Figure 22.5.1

This figure shows an undirected weighted graph with nodes labelled 0 to 4. Edges 0–1 and 2–4 have weight 20. Edges 1–2, 2–3, 3–4 and 3–0 have weight 10. Edges 0–4 and 1–3 have weight 5. Edges 0–2 and 1–4 have infinite weight.

31.13.4 Generate subsets

Figure 22.6.1

This figure shows a complete binary tree with four levels. Each node has a set and a sequence. The root node, in level 1, has the empty set and sequence (1,2,3). Its children, the nodes in level 2, have sequence (2,3). The left child has set {1} whereas the right child has the empty set. The four nodes in level 3 all have sequence (3). From left to right the sets are: {1,2}, {1}, {2} and

empty. The eight leaves in level 4 all have the empty sequence. Their sets are all the subsets of {1,2,3}.

31.14 Dynamic Programming

31.14.1 Fibonacci

Figure 23.1.1

This figure shows a binary tree with each node labelled $\text{fib}(n)$, for some n from 1 to 6. The left child of $\text{fib}(n)$ is $\text{fib}(n-1)$ and the right child is $\text{fib}(n-2)$. All leaves are labelled $\text{fib}(2)$ or $\text{fib}(1)$. The root is $\text{fib}(6)$ with left child $\text{fib}(5)$, which in turn has left child $\text{fib}(4)$, which in turn has left child $\text{fib}(3)$, which has children $\text{fib}(2)$ and $\text{fib}(1)$. $\text{fib}(4)$ has right child $\text{fib}(2)$. $\text{fib}(5)$ has a right child also labelled $\text{fib}(3)$, which in turn has again two children $\text{fib}(2)$ and $\text{fib}(1)$. The right child of root $\text{fib}(6)$ is another node labelled $\text{fib}(4)$, again with children $\text{fib}(3)$ and $\text{fib}(2)$, where $\text{fib}(3)$ has children $\text{fib}(2)$ and $\text{fib}(1)$.

Figure 23.1.2

This figure shows the same binary tree as before but now every right child has no further children. The root is $\text{fib}(6)$ with left child $\text{fib}(5)$ and right child $\text{fib}(4)$. $\text{fib}(5)$ has a left child also labelled $\text{fib}(4)$ and right child $\text{fib}(3)$. The $\text{fib}(4)$ child of $\text{fib}(5)$ has left child $\text{fib}(3)$ and right child $\text{fib}(2)$. The $\text{fib}(3)$ child of $\text{fib}(4)$ has children $\text{fib}(2)$ and $\text{fib}(1)$.

Figure 23.1.3

This figure shows a graph with six nodes labelled $\text{fib}(1)$ to $\text{fib}(6)$, from left to right. There are eight directed edges, all pointing left to right. The edges are: from $\text{fib}(1)$ to $\text{fib}(3)$; from $\text{fib}(2)$ to $\text{fib}(3)$ and $\text{fib}(4)$; from $\text{fib}(3)$ to $\text{fib}(4)$ and $\text{fib}(5)$; from $\text{fib}(4)$ to $\text{fib}(5)$ and $\text{fib}(6)$; from $\text{fib}(5)$ to $\text{fib}(6)$.

31.14.2 Longest common subsequence

Figure 23.2.1

This figure shows a graph with nodes labelled $\text{lcs}(y, n)$, where y is a substring of ‘yes’ and n is a substring of ‘no’. There’s an edge from $\text{lcs}(y_1, n_1)$ to $\text{lcs}(y_2, n_2)$ if the solution to $\text{lcs}(y_2, n_2)$ depends on the solution for $\text{lcs}(y_1, n_1)$. For each node, the node to its left and below skips the first letter of the left string, and the node to its right and above skips the first letter of the right string. The node at the top is $\text{lcs}(\text{‘yes’}, \text{‘no’})$. It has two incoming edges from $\text{lcs}(\text{‘es’}, \text{‘no’})$ to the left and $\text{lcs}(\text{‘yes’}, \text{‘o’})$ to the right. Node $\text{lcs}(\text{‘es’}, \text{‘no’})$ has also two incoming edges, from left node $\text{lcs}(\text{‘s’}, \text{‘no’})$ and right node $\text{lcs}(\text{‘es’}, \text{‘o’})$. Node $\text{lcs}(\text{‘s’}, \text{‘no’})$ has incoming edges from $\text{lcs}(\text{‘’}, \text{‘no’})$ to the left and $\text{lcs}(\text{‘s’}, \text{‘o’})$ to the right. Node $\text{lcs}(\text{‘s’}, \text{‘o’})$ has incoming edges from $\text{lcs}(\text{‘’}, \text{‘o’})$ and $\text{lcs}(\text{‘s’}, \text{‘’})$. Node $\text{lcs}(\text{‘es’}, \text{‘o’})$ has incoming edges from node $\text{lcs}(\text{‘s’}, \text{‘o’})$ and $\text{lcs}(\text{‘es’}, \text{‘’})$. Node $\text{lcs}(\text{‘yes’}, \text{‘o’})$ has incoming edges from $\text{lcs}(\text{‘es’}, \text{‘o’})$ and $\text{lcs}(\text{‘yes’}, \text{‘’})$.

Figure 23.2.2

This figure shows a table with five columns numbered 0 to 4 from left to right and five rows numbered 0 to 4 from top to bottom. Each table entry is of the form $\text{lcs}(l, r)$, where l is a substring of ‘soho’ and r is a substring of ‘ohio’. The cells in row 0 are, from left to right, $\text{lcs}(\text{'soho'}, \text{'ohio'})$, $\text{lcs}(\text{'soho'}, \text{'hio'})$, $\text{lcs}(\text{'soho'}, \text{'io'})$, $\text{lcs}(\text{'soho'}, \text{'o'})$ and $\text{lcs}(\text{'soho'}, \text{''})$. The cells in row 1 are $\text{lcs}(\text{'oho'}, \text{'ohio'})$, $\text{lcs}(\text{'oho'}, \text{'hio'})$, $\text{lcs}(\text{'oho'}, \text{'io'})$, $\text{lcs}(\text{'oho'}, \text{'o'})$ and empty. The cells in row 2 are empty, $\text{lcs}(\text{'ho'}, \text{'hio'})$, $\text{lcs}(\text{'ho'}, \text{'io'})$, $\text{lcs}(\text{'ho'}, \text{'o'})$ and $\text{lcs}(\text{'ho'}, \text{''})$. The cells in row 3 are empty, empty, $\text{lcs}(\text{'o'}, \text{'io'})$, $\text{lcs}(\text{'o'}, \text{'o'})$ and empty. The cells in row 4 are empty, empty, $\text{lcs}(\text{''}, \text{'io'})$, empty and $\text{lcs}(\text{''}, \text{''})$. The figure has arrows between cells showing their dependencies. There are four diagonal arrows from one cell to the one left and above it. They are: from $\text{lcs}(\text{''}, \text{''})$ in row 4 column 4 to $\text{lcs}(\text{'o'}, \text{'o'})$ in row 3, column 3; from $\text{lcs}(\text{'ho'}, \text{''})$ in row 2 column 4 to $\text{lcs}(\text{'oho'}, \text{'o'})$ in row 1, column 3; from $\text{lcs}(\text{'o'}, \text{'io'})$ in row 3 column 2 to $\text{lcs}(\text{'ho'}, \text{'hio'})$ in row 2, column 1, which in turn has a diagonal arrow to $\text{lcs}(\text{'oho'}, \text{'ohio'})$ in row 1, column 0. All other non-empty cells, except those in the last row or column, have two incoming edges from their left and bottom neighbour.

Figure 23.2.3

This figure shows the same 5 by 5 table as the previous figure, but the empty cells are now filled and their dependencies have been added. Row 4: $\text{lcs}(\text{''}, \text{'ohio'})$ in column 0, $\text{lcs}(\text{''}, \text{'hio'})$ in column 1 and $\text{lcs}(\text{''}, \text{'o'})$ in column 3. Row 3: $\text{lcs}(\text{'o'}, \text{'ohio'})$ in column 0 with a diagonal arrow from $\text{lcs}(\text{''}, \text{'hio'})$; $\text{lcs}(\text{'o'}, \text{'hio'})$ in column 1 with arrows from $\text{lcs}(\text{''}, \text{'hio'})$ below and $\text{lcs}(\text{'o'}, \text{'io'})$ to the right; $\text{lcs}(\text{'o'}, \text{''})$ in column 4. Row 2: $\text{lcs}(\text{'ho'}, \text{'ohio'})$ in column 0 with arrows from $\text{lcs}(\text{'o'}, \text{'ohio'})$ below and $\text{lcs}(\text{'ho'}, \text{'hio'})$ to the right. Row 1: $\text{lcs}(\text{'oho'}, \text{''})$ in column 4.

31.14.3 Knapsack

Figure 23.3.1

This figure shows a digraph. Each node is a pair of integers from 0 to 4. The graph is laid out in five levels. Pairs of the form (n, c) are in level n . Pairs starting with 4 are leaves, because all items were considered. Pairs ending with 0 are leaves because there’s no capacity left for more items. The node in the top level is $(0, 4)$. It has incoming arrows from $(1, 4)$ on the left and $(1, 3)$ on the right. Node $(1, 4)$ has incoming arrows from $(2, 4)$ on the left and $(2, 3)$ on the right. Node $(1, 3)$ has incoming arrows from $(2, 3)$ on the left and $(2, 2)$ on the right. Node $(2, 4)$ has incoming arrows from $(3, 4)$ on the left and $(3, 1)$ on the right. Node $(2, 3)$ has incoming arrows from $(3, 3)$ on the left and $(3, 0)$ on the right. Node $(2, 2)$ has one incoming arrow from $(3, 2)$ on the left. Node $(3, 4)$ has incoming arrows from $(4, 4)$ on the left and $(4, 0)$ on the right. Node $(3, 1)$ has one incoming arrow from $(4, 1)$ on the left. Node $(3, 3)$ has one incoming arrow from $(4, 3)$ on the left. Node $(3, 2)$ has one incoming arrow from $(4, 2)$ on the left.

31.15 Practice 3

31.15.1 Extra staff

Figure 24.2.1

This figure shows a directed acyclic graph with 9 nodes, labelled 0 to 8. Node 1 has edges to nodes 0 and 6. Node 0 has edges to nodes 5 and 6. Node 5 has an edge to node 8, which in turn has edges to nodes 6 and 7. Node 4 has edges to nodes 2 and 3.

31.15.2 Borrow a book**Figure 24.3.1**

This figure shows a directed graph with 21 nodes. They are laid out in three rows of seven nodes each. Each node is labelled with a letter–weekday pair. Left to right in each row, the weekdays are Mon, Tue, Wed, Thu, Fri, Sat and Sun. From top to bottom, the letters in each row are A, B and C. Each node has an edge to the node to its right in the same row, e.g. from A, Mon to A, Tue and from B, Thu to B, Fri. In each row there's an edge from the right-most node to the left-most node, e.g. from A, Sun to A, Mon and from B, Sun to B, Mon. All these edges have weight 1. In addition there are three pairs of edges across rows. These six edges have weight 0. There's a pair of edges from B, Tue to C, Tue and back to C, Tue. The second pair goes from A, Wed to B, Wed and back to A, Wed. The third pair goes from A, Sat to C, Sat and back to A, Sat.

31.15.3 Levenshtein distance**Figure 24.4.1**

This figure shows a DAG where the nodes are of the form $\text{edit}(\text{left}, \text{right})$. The root node is $\text{edit}(\text{'rate'}, \text{'grate'})$. Its left child is $\text{edit}(\text{'ate'}, \text{'grate'})$, its middle child is $\text{edit}(\text{'ate'}, \text{'rate'})$ and its right child is $\text{edit}(\text{'rate'}, \text{'rate'})$. Node $\text{edit}(\text{'ate'}, \text{'grate'})$ also has three children. From left to right they are: $\text{edit}(\text{'te'}, \text{'grate'})$, $\text{edit}(\text{'te'}, \text{'rate'})$ and $\text{edit}(\text{'ate'}, \text{'rate'})$. The latter is also the middle child of the root node. Node $\text{edit}(\text{'te'}, \text{'grate'})$ has right child $\text{edit}(\text{'te'}, \text{'rate'})$, which is also the middle child of $\text{edit}(\text{'ate'}, \text{'grate'})$ and the left child of $\text{edit}(\text{'ate'}, \text{'rate'})$. The right child of the root, $\text{edit}(\text{'rate'}, \text{'rate'})$, has a single, middle child: $\text{edit}(\text{'ate'}, \text{'ate'})$.

31.15.4 Higher and higher**Figure 24.5.1**

This figure shows a grid with two rows and three columns. From left to right, the top row has numbers 10, 7 and 10, and the bottom row has numbers 3, 90 and 82.

Figure 24.5.2

The grid in this figure has three rows and three columns. From left to right, the numbers are 6, 7 and 8 in the top row, 6, 5 and 4 in the middle row, and 3, 2 and 1 in the bottom row.

31.16 Complexity classes

31.16.1 The P and NP classes

Figure 26.2.1

This figure shows two diagrams. The left-hand diagram is for the case $P = NP$. It consists of a rectangle with one smaller rectangle inside it. The outer rectangle is divided in two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The inner rectangle is the outline of the top left-hand cell, which is labelled with P and NP , and represents the tractable decision problems. The right-hand diagram is for the case where P is included in NP . It also consists of an outer rectangle and an inner rectangle. The outer rectangle has the same two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The inner rectangle includes the top left-hand cell of tractable decision problems, labelled P , and part of the bottom left-hand cell of intractable decision problems. The boundary between tractable and intractable is marked with a dashed line. The intractable part of the rectangle is labelled with NP .

31.16.2 Reductions

Figure 26.3.1

In this figure, a large box labelled ‘Unsolved’ has three small boxes inside, numbered 1 to 3 from left to right. Box 1, the left inner box, is labelled ‘IT’. Box 2, the middle inner box, is labelled ‘Solved’. Box 3, the right inner box, is labelled ‘OT’. There are four arrows, all pointing to the right. An arrow labelled ‘inputs’ goes from outside the large ‘Unsolved’ box to the IT box. A second arrow, also labelled ‘inputs’, goes from the ‘IT’ to the ‘Solved’ box. A third arrow, labelled ‘output’, goes from the ‘Solved’ to the ‘OT’ box. A fourth arrow, also labelled ‘output’, goes from the ‘OT’ box to outside the large ‘Unsolved’ box.

Figure 26.3.2

In this figure, a large box labelled ‘Unsolved: interval scheduling’ has three small boxes inside, numbered 1 to 3 from left to right. Box 1, the left inner box, is labelled ‘IT’. Box 2, the middle inner box, is labelled ‘Solved: maximal independent set’. Box 3, the right inner box, is labelled ‘OT’. There are five arrows, all pointing to the right. One arrow labelled ‘intervals’ goes from outside the large ‘Unsolved’ box to the ‘IT’ box. Two arrows, labelled ‘items’ and ‘incompatible’, go from the ‘IT’ to the ‘Solved’ box. One arrow labelled ‘compatible’ goes from the ‘Solved’ to the ‘OT’ box. One arrow labelled ‘schedule’ goes from the ‘OT’ box to outside the large ‘Unsolved’ box.

31.16.3 Problem hardness

Figure 26.4.1

This figure has two box diagrams, one above the other. In the top diagram, a large box labelled ‘Problem A’ has three smaller boxes inside. The left inner box is labelled ‘IT A’. The middle inner box is labelled ‘Problem B’. The right inner box is labelled ‘OT B’. The ‘Problem B’ box has three boxes inside it, labelled from left to right as ‘IT B’, ‘Problem C’ and ‘OT C’. There are

six arrows, all pointing to the right. The arrow labelled ‘I A’ goes from outside the ‘Problem A’ box to the ‘IT A’ box. The arrow labelled ‘I B’ goes from the ‘IT A’ to the ‘IT B’ box. The arrow labelled ‘I C’ goes from the ‘IT B’ to the ‘Problem C’ box. The arrow labelled ‘O C’ goes from the ‘Problem C’ box to the ‘OT C’ box. The arrow labelled ‘O B’ goes from the ‘OT C’ box to the ‘OT B’ box. The arrow labelled ‘O A’ goes from the ‘OT B’ box to outside the ‘Problem A’ box. In the bottom diagram, a large box labelled ‘Problem A’ has three smaller boxes inside. The left inner box is labelled ‘IT A + IT B’. The middle inner box is labelled ‘Problem C’. The right inner box is labelled ‘OT C + OT B’. There are four arrows, all pointing to the right. The arrow labelled ‘I A’ goes from outside the ‘Problem A’ box to the ‘IT A + IT B’ box. The arrow labelled ‘I C’ goes from the ‘IT A + IT B’ box to the ‘Problem C’ box. The arrow labelled ‘O C’ goes from the ‘Problem C’ box to the ‘OT C + OT B’ box. The arrow labelled ‘O A’ goes from the ‘OT C + OT B’ box to outside the ‘Problem A’ box.

Figure 26.4.2

This figure shows two diagrams. The left-hand diagram is for the case $P = NP$. It consists of a rectangle with two smaller rectangles inside it. The outer rectangle is divided in two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The first inner rectangle is the outline of the top left-hand cell, which is labelled with P and NP, and represents the tractable decision problems. The second inner rectangle, labelled NP-hard, is in the centre of the outer rectangle, overlapping part of each of the four cells. The overlap between the two inner rectangles is labelled NP-complete. The right-hand diagram is for the case where P is included in NP . It also consists of an outer rectangle and two inner rectangles. The outer rectangle has the same two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The first inner rectangle includes the top left-hand cell of tractable decision problems, labelled P, and part of the bottom left-hand cell of intractable decision problems. The boundary between tractable and intractable is marked with a dashed line. The intractable part of the rectangle is labelled with NP. The second inner rectangle, labelled NP-hard, covers part of both columns in the bottom row, so the intractable decision and non-decision problems. The overlap of both inner rectangles is again labelled NP-complete.

31.16.4 Summary

Figure 26.6.1

This figure shows two diagrams. The left-hand diagram is for the case $P = NP$. It consists of a rectangle with two smaller rectangles inside it. The outer rectangle is divided in two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The first inner rectangle is the outline of the top left-hand cell, which is labelled with P and NP, and represents the tractable decision problems. The second inner rectangle, labelled NP-hard, is in the centre of the outer rectangle, overlapping part of each of the four cells. The overlap between the two inner rectangles is labelled NP-complete. The right-hand diagram is for the case where P is included in NP . It also consists of an outer rectangle and two inner rectangles. The outer rectangle has the same two columns, labelled decision and non-decision, and two rows, labelled tractable and intractable. The first inner rectangle includes the top left-hand cell of tractable decision problems, labelled P, and part of the bottom left-hand cell of intractable decision problems. The boundary between tractable and intractable is marked with a dashed line. The intractable part of the rectangle is labelled with NP. The second inner rectangle, labelled NP-hard, covers part of both columns in the bottom row, so the intractable decision and non-decision problems. The overlap of both inner rectangles is again labelled NP-complete.

NP-hard, covers part of both columns in the bottom row, so the intractable decision and non-decision problems. The overlap of both inner rectangles is again labelled NP-complete.

31.17 Computability

31.17.1 Turing machine

Figure 27.1.1

This figure shows the first five cells of the tape. The top half of the figure shows them before processing, the bottom half after processing. Above each cell is the current state. Below each cell is the head movement, depicted as an arrow pointing left or right. In the top half, from left to right, the states, symbols and head movements are as follows. In the start state, the head is over a blank and then moves right. The machine is now in the even state, over a 1, and moves right. The machine is now in the odd state, over a 0, and moves right. The machine is now in the even state, over a blank. The bottom half of the diagram must be read right to left. The machines replaces the blank with a zero and moves left. The machine is now in the back state, over a 1, and moves left. The machine is now in the back state, over a 0, and moves left. The machine is now in the back state, over a 1, and moves left. The machine is now in the back state, over a blank, and doesn't move.

31.17.2 The Church–Turing thesis

Figure 27.2.1

This figure shows three rows of small diagrams, each showing the content of the first four cells of the tape. Each diagram also indicates the current state and the current head position. In the first row, from left to right, there are diagrams: 1 The state is start, the head is on cell 1, the tape is a, a, blank, blank. 2 The state is up, the head is on cell 2, the tape is X, a, blank, blank. 3 The state is up, the head is on cell 3, the tape is X, a, blank, blank. 4 The state is back, the head is on cell 2, the tape is X, a, 1, blank. 5 The state is back, the head is on cell 1, the tape is X, a, 1, blank. The second row also has five diagrams: 1 The state is start, the head is on cell 2, the tape is a, a, 1, blank. 2 The state is up, the head is on cell 3, the tape is a, X, 1, blank. 3 The state is up, the head is on cell 4, the tape is a, X, 0, blank. 4 The state is back, the head is on cell 3, the tape is a, X, 0, 1. 5 The state is back, the head is on cell 2, the tape is a, X, 0, 1. The third row has a single diagram: the state is start, the head is on cell 3, the tape is a, a, 0, 1.

31.17.3 Undecidability

Figure 27.4.1

This figure shows two diagrams. The left-hand diagram is for the case P = NP. It consists of a rectangle with three smaller rectangles inside it. The outer rectangle is divided in two columns, labelled decision and non-decision, and three rows, labelled tractable, intractable and not computable. The first inner rectangle is the outline of the top left-hand cell, which is labelled with P and NP, and represents the tractable decision problems. The second inner rectangle is the outline of the bottom left-hand cell, which is labelled with undecidable, and represents the non-computable decision problems. The third inner rectangle, labelled NP-hard, is in the centre

of the outer rectangle, overlapping part of each of the six cells. The overlap between the first and third inner rectangles is labelled NP-complete. The right-hand diagram is for the case where P is included in NP. It also consists of an outer rectangle and three inner rectangles. The outer rectangle has the same two columns, labelled decision and non-decision, and three rows, labelled tractable, intractable and not computable. The first inner rectangle includes the top left-hand cell of tractable decision problems, labelled P, and part of the middle left-hand cell of intractable decision problems. The boundary between tractable and intractable is marked with a dashed line. The intractable part of the rectangle is labelled with NP. The second inner rectangle is the outline of the bottom left-hand cell, which is labelled with undecidable, and represents the non-computable decision problems. The third inner rectangle, labelled NP-hard, covers part of both columns in the middle and bottom rows, so the intractable and the non-computable decision and non-decision problems. The overlap of the first and third inner rectangles is labelled NP-complete.

31.17.4 Summary

Figure 27.5.1

This figure shows two diagrams. The left-hand diagram is for the case $P = NP$. It consists of a rectangle with three smaller rectangles inside it. The outer rectangle is divided in two columns, labelled decision and non-decision, and three rows, labelled tractable, intractable and not computable. The first inner rectangle is the outline of the top left-hand cell, which is labelled with P and NP, and represents the tractable decision problems. The second inner rectangle is the outline of the bottom left-hand cell, which is labelled with undecidable, and represents the non-computable decision problems. The third inner rectangle, labelled NP-hard, is in the centre of the outer rectangle, overlapping part of each of the six cells. The overlap between the first and third inner rectangles is labelled NP-complete. The right-hand diagram is for the case where P is included in NP. It also consists of an outer rectangle and three inner rectangles. The outer rectangle has the same two columns, labelled decision and non-decision, and three rows, labelled tractable, intractable and not computable. The first inner rectangle includes the top left-hand cell of tractable decision problems, labelled P, and part of the middle left-hand cell of intractable decision problems. The boundary between tractable and intractable is marked with a dashed line. The intractable part of the rectangle is labelled with NP. The second inner rectangle is the outline of the bottom left-hand cell, which is labelled with undecidable, and represents the non-computable decision problems. The third inner rectangle, labelled NP-hard, covers part of both columns in the middle and bottom rows, so the intractable and the non-computable decision and non-decision problems. The overlap of the first and third inner rectangles is labelled NP-complete.