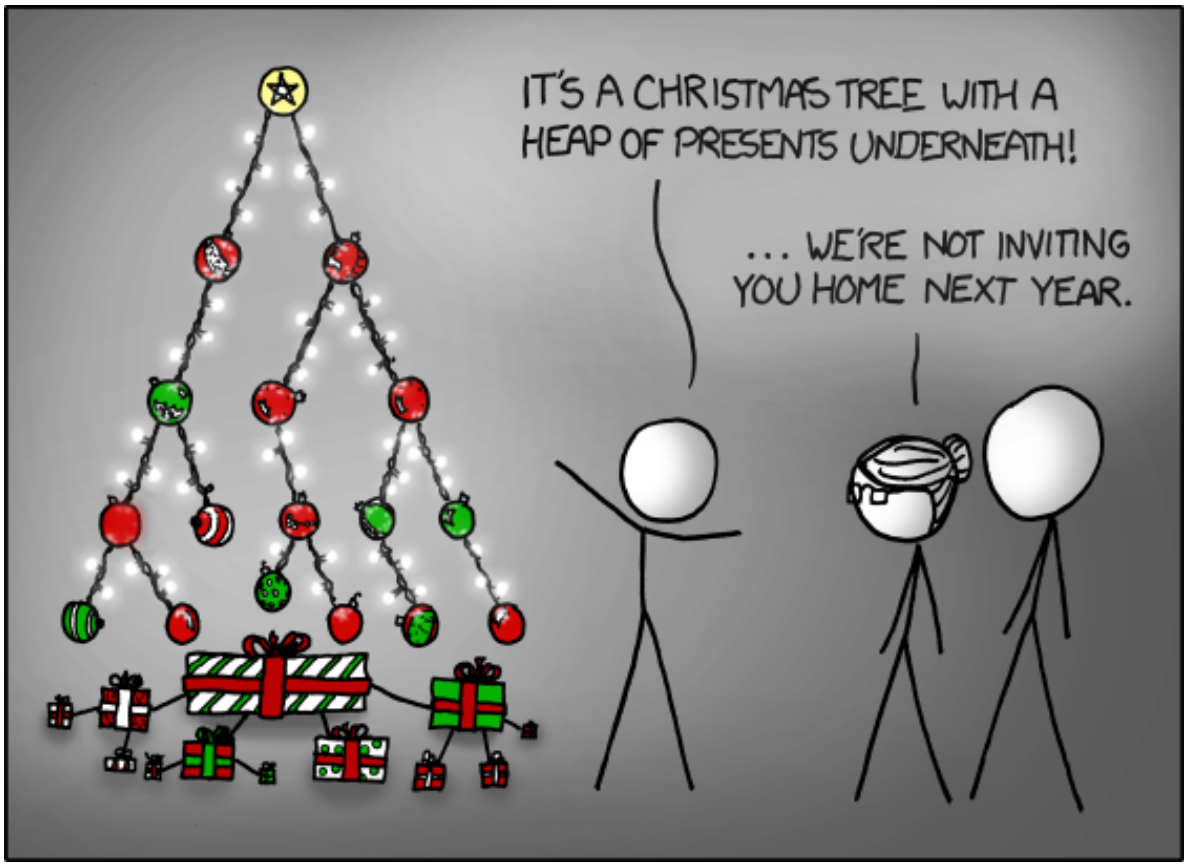
Advanced Data Structures



Jim Newton

July 23, 2026

Contents

Chapter 0

Introduction

Course Objectives

- Implement fundamental data structures — linked lists, stacks, binary search trees, heaps, and graphs — from scratch in Python.
- Apply graph algorithms including depth-first search, breadth-first search, topological sort, and shortest-path algorithms (Bellman-Ford, Dijkstra) to solve concrete problems.
- Represent computational problems (expression evaluation, text compression, state machines, puzzle solving) as data structures and implement the corresponding algorithms.
- Recognise how mathematical definitions (formal graph theory, regular languages, tree isomorphism) correspond directly to Python class hierarchies and recursive functions.
- Write systematic unit tests for data structure implementations and reason about correctness from a specification.

The course Algorithmics and Data Structures offers an introduction to several fundamental data structures such as stacks, trees, heaps, and graphs. Rather than concentrating on theoretical aspects the course emphasises programming using these data structures in Python.

0.1 Evaluation

Your grade will be determined as follows.

- 35% Homework via Forge/Intranet
- 25% Group Project
- 10% Quizzes
- 30% Final-Term Exam

0.2 How to Read This Document

Throughout this document, margin annotations and colored boxes serve as navigation aids.

Margin annotations. Icons appearing in the margin signal the nature of the adjacent content.

- **Essential** — a core concept you must understand and be able to apply on the exam.
- **Recall** — a result or technique introduced elsewhere; look it up if needed.
- **Warning** — a common mistake or a subtle point requiring extra care.
- **Enrichment** — optional deeper material for the curious reader; not assessed.

Colored boxes. Different types of content are distinguished by background color.

Chapter Objectives — the skills and understanding you are expected to have by the end of the chapter.

Activity — an in-class exercise or hands-on task to complete with your group.

Definition — a precise mathematical meaning assigned to a term.

Claim — a mathematical statement that follows from the definitions and is used in a proof.

Proof — a logical argument establishing a claim.

Listing — a Python code extract for reference or study; the margin icon marks later uses of a pattern introduced earlier.

0.3 Syllabus Spring Semester

- Chap 1: Linked Lists
  - Compared to Arrays
  - Mutability vs Immutability
  - Reverse Polish Notation
- Chap 2: Bartosz VM
  - Instruction set
  - Group Project
  - Debugging
- Chap 3: Binary Trees
  - Binary Search Trees
  - Priority Queue
  - Huffman Encoding
- Chap 4: Graphs
  - Directed/Undirected Graphs
  - Adjacency List
  - DFS
  - Topological sort
- Chap 5: Abstract Syntax Trees
  - Expression Evaluators
  - Arithmetic Expressions
  - Boolean Expressions
  - Serialize expression as infix, prefix (lisp), or postfix (RPN)
- Chap 8: BFS
  - BFS
  - Solving Rubik's Cube

0.4 Syllabus Fall Semester

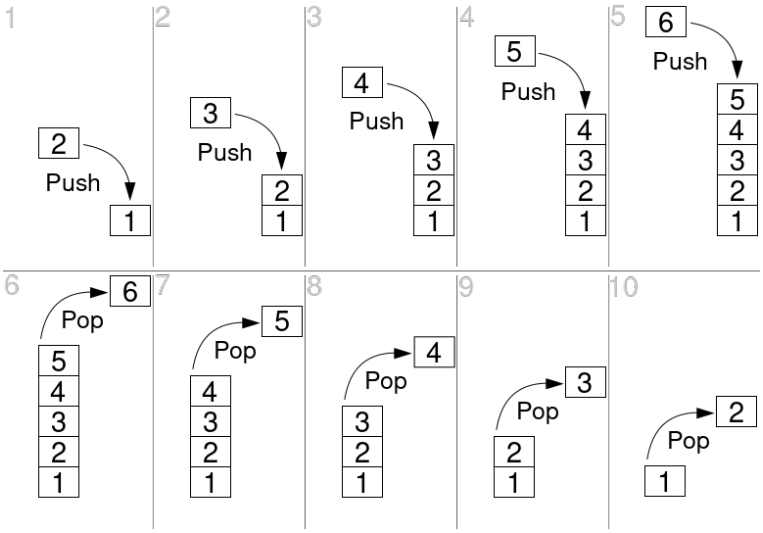
- Chap 6: Finite Automata
  - As weighted Graph
  - Regular Expressions
  - Determinism
- Chap 7: Graph Isomorphism
  - Adjacency Matrix
  - Zero Knowledge Proof
- Chap 9: Shortest Path Graph Search
  - Bellman Ford
  - Dijkstra

Chapter 1

Stacks

Chapter Objectives

- Implement a stack using Python's built-in `list` and explain why `append` and `pop` give LIFO behaviour.
- Implement a stack using a fixed-size array with a fill-pointer (`sp`) and extend it to double in capacity when full.
- Identify the aliasing and index-access problems that make mutable stacks unsafe in the presence of shared references or untrusted functions.
- Implement an immutable stack as nested Python tuples, where `None` is the empty stack and each node is a `(value, rest)` pair.
- Explain why immutable stacks allow safe sharing of tails and cannot be corrupted by a function that receives a reference to the stack.
- Write a Python generator that iterates over an immutable stack in LIFO order without allocating an intermediate list.
- Trace the stack state step-by-step through an RPN calculator computation, identifying which keystrokes push, duplicate, and apply operators.



A stack is a data structure with a simply API which implements a LIFO (last in, first out) collection. The basic operations let you *push* on a new item, and *pop* off the most recent item. The *pop* operation (depending on the implementation) may give you access to the item discarded, or it may be necessary to *read* the top item before discarding it.

Some stack implementations allow other more complex operations such as *duplicate* the top item, *swap* the top two items, or *peek* into the stack up to a given depth.

1.1 Python `list` as Stack

The built-in `list` data structure in Python may be used as a stack. The *push* operation is the `append` method; the *pop* operation is the `pop` method.

The methods `append` and `pop` modify the stack.

```
>>>   st = []
>>>   st.append(12)
>>>   st.append(42)
>>>   st.append(-4)
>>>   print(st)
 [12, 42, -4]

>>> st.pop()
 -4

>>> st.append(100)
>>> st.append(200)
>>>   print(st)
 [12, 42, 100, 200]

>>> a = st.pop()
>>> b = st.pop()
>>> print(a)
 200

>>> print(b)
 100

>>> print(st)
 [12, 42]
```

As with all destructive functions in Python, it is difficult (sometimes impossible) to know the consequence of using the `append` and `pop` methods. As an example, consider the following code.

```
  >>> st1 = []
  >>> st2 = st1
  >>> pair = (st1, st2)
  >>> st1.append(100)
  >>> st2.append(200)
  >>> st1.append(300)
  >>> st2.pop()
  >>> print(st1)
 [100, 200]

  >>> print(st2)
 [100, 200]

  >>> print(pair)
 ([100, 200], [100, 200])
```

In the case of the previous example, it is straightforward to reason about the the side effects. `st1` and `st2` reference the same list allocated in memory, so `st1.append`, and `st2.append` operate on the same exact

stack. There is one stack with multiple references to it. We have two variables, `st1` and `st2`, referring to the same stack, but we also have a composite data structure `pair` which contains two additional references. So `st1.append(...)`, `st2.append(...)`, and `st2.pop()` have also modified `pair`.

However, consider the following function.

```
def f(st, x, y, z):
  st.append(x)
  st.append(y+z)
```

In this case, the function `f` has no way of knowing how many references the program has made to the stack.

1.2 Array as Stack

The fact that Python has the `append` and `pop` methods which treat `list` as a stack somewhat obviate the need to look at another implementation of array as stack. However, since many other programming languages do not have this feature, you might ask how to implement a stack.

One way to implement a stack with a fixed size array, and a so-called *fill-pointer*. The *push* and *pop* instructions modify this *fill-pointer*. Such an implementation needs to know the beginning and end of the array. In Python, the `len` function returns the length of the list (array), but in some languages, the language itself does not know the end; rather it is left as a subtle task for the programmer to remember where the end of the array is, and to never write past the end of the array.

The Bartosz machine in Chapter 2 uses this approach.

The following is an example. The variable `sp`, *stack-pointer*, serves two purposes: it indicates how many items are currently on the stack, and it indicates the index of the next item to be pushed.

An alternative implementation would have `sp` reference the most recently pushed item, *i.e.*, the top of the stack. In such a case, the initial value of `sp` would need to be `-1`.

```
size = 10
sp = 0
st = [None] * size

def st_push(v):
  global st, sp

  assert sp < size, "stack overflow"
  st[sp] = v
  sp = sp + 1

def st_pop():
  global st, sp

  assert sp > 0, "stack underflow"
  sp = sp - 1
  return st[sp]
```

The `st_push` function pushes a new element by modifying the array at index `sp`, but first makes sure we have enough space on the stack. Of course an alternative implementation would simply allocate more space rather than failing if an attempt were made to push too many elements onto the stack.

```
def st_push(v):
  global st, sp, size

  if sp >= size:
    st = st + [None] * size
    size = 2 * size

  st[sp] = v
  sp = sp + 1
```

Here is an example of how this stack implementation might be used.

```
st_push(12)
st_push(42)
st_push(100)
print(st)
--> [12, 42, 100, None, None, None, None, None, None, None]

print(sp)
--> 3
```

The `st_pop` function returns the element at the top of the stack, i.e. one behind `sp`, but first makes sure the stack is non-empty.

```
x = st_pop()
print(x)
--> 100

print(sp)
--> 2

print(st)
--> [12, 42, 100, None, None, None, None, None, None, None]
```

Notice, that the `st_pop` function does not clean up the stack. Even though we popped off `100`, the stack still contains this value. However, `sp` has value `2` indicating that there are only 2 items on the stack.

```
st_push(111)
print(sp)
--> 3

print(st)
--> [12, 42, 111, None, None, None, None, None, None, None]
```

Of course we could implement the `st_pop` function to discard elements from the stack, once they have been popped.

```
def st_pop():
  global st, sp

  assert sp > 0, "stack underflow"
  sp = sp - 1
  v = st[sp]
  st[sp] = None
  return v
```

Notice with this implementation, the stack is global. The code (as written) does not support multiple independent stacks. You could fix this problem by using Object Orientation. *I.e.*, create a `Stack` class with instance variables, `sp` and `st`, and the functions `st_push` and `st_pop` could become the methods `push` and `pop` — the `st_` being no longer necessary.

We leave such an OO implementation as an exercise for the student.

1.3 Limitations of the Mutable Stack

A disadvantage of the implementations of stack in Sections 1.1 and 1.2, in both cases, is that the data on the stack is not protected in any way. Anyone with access to the underlying list can intentionally or accidentally modify it. For example, from Section 1.1, imagine that `st`, is a local variable which is being treated as a stack, how can you write the following function correctly?

```
def op(f,st):
  st.append(True)
  v = f(st)
  st.pop() #pop off the True we pushed above
  return v
```

The problem is that we do not know whether the function `f` intentionally or accidentally modified the stack. How can we know that the true value `True` is still on the stack, or that it hasn't been duplicated? We could of course copy `st` before calling `f`. However, this would be inefficient because *usually* the function `f` should not modify the stack.

In general, Python programmers, ignore this issue and rely on the hope that different parts of the program cooperate, and that different programmers read, follow, and maintain the docstrings.

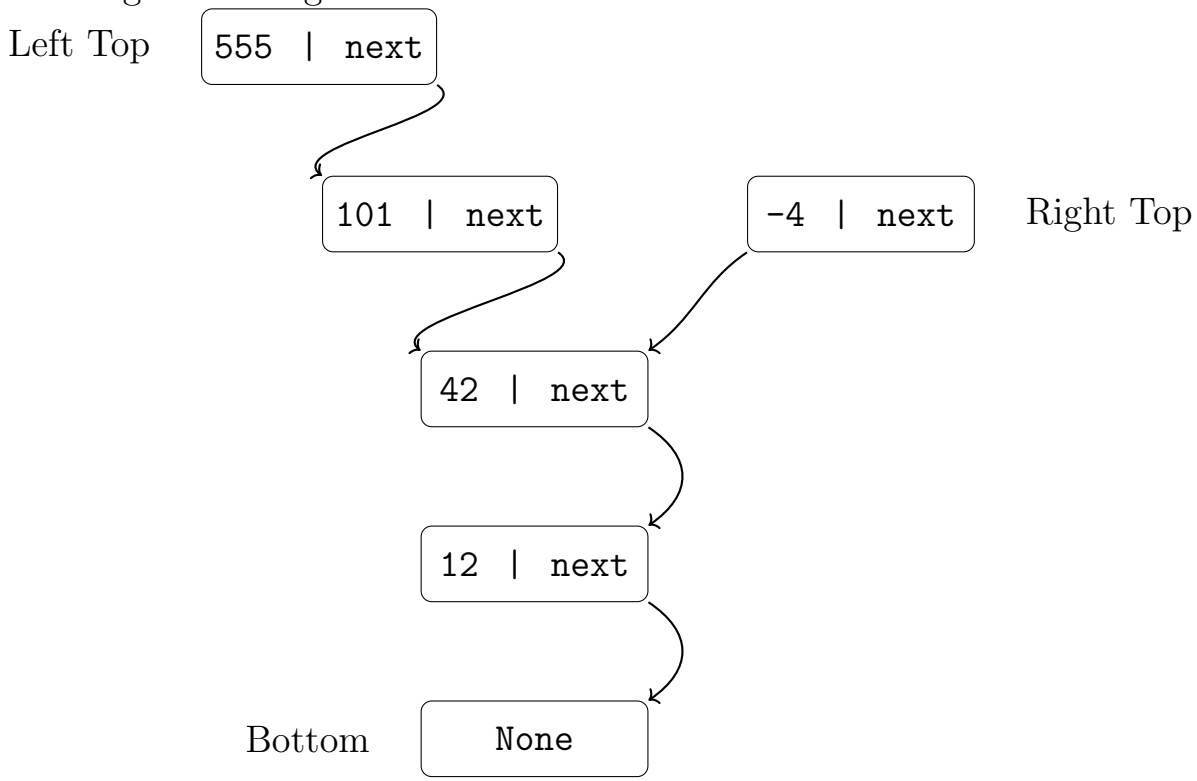
Another problem with the previous implementations is that because the stack is implemented as a `list` which is an array in Python, the entire stack can be accessed by index: `st[0]`, `st[1]`, ... `st[n]`. This is a problem because the concept of stack is that only the top of the stack is immediately accessible.

Yet another limitation of the previous implementations is two different stacks cannot share a tail. If we want two stacks which agree about 1000 items, but differ by the most recent item, then we must copy the entire stack.

We can do a bit better in this regard, by relying on the Python feature that tuples are immutable.

1.4 Immutable Stack

In this section we look at an immutable implementation of a stack. The following is an image of two stacks with a shared tail.



We would like to create such a structure as follows using the function `ims_push` (immutable-stack-push). Notice, that the bottom node is not a *pair* but is the `None` object, representing an empty stack. Otherwise, each stack entry is implemented by a pair containing the stacked item, and a reference to another stack.

There are several things to notice about this approach.

1. `None` represents the empty stack
2. To push an item, we need some existing stack (or the empty stack) as first argument to `ims_push`.
3. The `ims_push` function does not modify an existing stack, but rather returns a new one.

4. The `ims_push` function does not invalidate the stack; the previous stack is still perfectly valid and can be reused.

5. A call to an *unsafe* function, `f`, cannot destroy the stack: `f(st)`.

The following is example code for generate the two stacks shown above.

```
st = ims_push(None,12)
st = ims_push(st, 42)
right_st = ims_push(st, -4)

_, st = ims_pop(right_st)
st = ims_push(st, 101)
left_st = ims_push(st, 555)
```

The following is a sample implementation of the *push* and *pop* functions. The `ims_push` function returns a new stack, given an item and an old (possibly empty) stack. The `ims_pop` function returns a tuple (*i.e.*, two values): the item being popped, and the stack after item has been popped. The caller is not required to *use* either of these return values, but most importantly, `ims_pop` does not in any way modify the existing stack. Unused values in Python are often indicated by using an underscore as variable: `_, st = ims_pop(right_st)`.

```
def ims_push(stack, value):
  return value, stack

def ims_pop(stack):
  assert stack is not None, "stack underflow"

  return stack
```

The student should notice that the Python code implementation for `ims_push` and `ims_pop` is extremely trivial—almost to the extent that the code is unnecessary. Indeed, we could easily implement the same thing using Python build-in features without much loss of readability. The advantage of using the accessor functions `ims_push` and `ims_pop` is that a person (even yourself when you look at the code a year later) should immediately realized that you are performing stack operations.

```
st = 12, None
st = 42, st
right_st = -4, st

st, _ = right_st
st = 101, st
left_st = 555, st
```

1.5 Iteration over a Stack

A limitation of the immutable implementation of the stack in Section 1.4, is that it does not behave well with the Python `for` command. In fact, an advantage of the list-based implementations we saw in Sections 1.1 and 1.2 is that they behave well with `for` because after all, the stack is a list. Here is a quick example. Notice, `for` traverses the stack from bottom to top rather than from top to bottom, which violates the LIFO (last-in first-out) behavior. Other than that, the behavior is pretty easy, and not surprising.

```
>>>  st = []
>>>  st.append(12)
>>>  st.append(42)
>>>  st.append(-4)
>>>  print(st)
 [12, 42, -4]

>>> for i in st:
      print(f"stack: {i}")
stack: 12
stack: 42
stack: -4
```

Now, what happens if we try to use `for` with th stack in Section 1.4?

```
>>>  st = 12, None
>>>  st = 42, st
>>>  right_st = -4, st
>>>  for i in st:
      print(f"stack: {i}")
stack: -4
stack: (42, (12, None))
```

This strange behavior of `for` can be understood because `st` is a tuple of length two. The first is `-4`; the second element is another stack,

in particular `(42, (12, None))`.

We present two ways to tackle this problem: first a simple way, and second, a more exotic way.

First we can convert a `ims` stack to a list as follows:

```
def ims_to_list(ims_st) -> list:
    computed = []
    while ims_st is not None:
        value, ims_st = ims_pop(ims_st)
        computed.append(value)
    return computed
```

Now, we can iterate not over the stack, but over the return value of `ims_to_list`. Notice that the LIFO behavior is implemented, the most recently pushed item is the first one printed. A disadvantage with this approach is that if the stack is large (*i.e.*, if the stack is deep), then we must allocate a large list simply in order to iterate across the stack.

```
>>>  st = 12, None
>>>  st = 42, st
>>>  st = -4, st
>>>  for i in ims_to_list(st):
      print(f"stack: {i}")
stack: -4
stack: 42
stack: 12
```

A second, more clever, alternative is to use a Python generator. To do this, we write a Python function that invokes `yield` rather than `return`.

```
def ims_to_generator(ims_st):
    while ims_st:
        value, ims_st = ims_pop(ims_st)
        yield value
```

The `yield` keyword indicates that when `for` asks for the next element, then `yield` will provide one, but most importantly, the function does not return; rather the function maintains its hidden internal state, and continues executing where it left off, the next time `for` asks for another element.

```
>>>   st = 12, None
>>>   st = 42, st
>>>   st = -4, st
>>>   for i in ims_to_generator(st):
       print(f"stack: {i}")
stack: -4
stack: 42
stack: 12
```

Again, the items are printed in LIFO order, the most recently pushed item, `-4` is printed first. The iteration finishes unceremoniously when the `None` at the bottom of the stack is reached.

The generator-based solution, `ims_to_generator`, fixes the problem created in `ims_to_list` in that no additional, intermediate, large list need be allocated. Rather, the generator simply iterates through the structure of the ims-stack.

1.5.1 Homework

There is a homework assignment `homework/stack.py`; the test case is found in `tests/stack_test.py`. The student should implement the functions which are incomplete in this file.

The homework uses the same immutable tuple-linked stack structure from Section 1.4, but with a different set of function names. This is deliberate: the same abstract data structure can be given many different APIs, and a key skill in this course is reading a docstring as a specification and implementing it—rather than mapping function names back to the chapter. The homework also asks you to implement several operations not discussed in the chapter: `swap`, `duplicate`, `reverse`, `map`, and `fold`. In each case, treat the docstring as your complete specification.

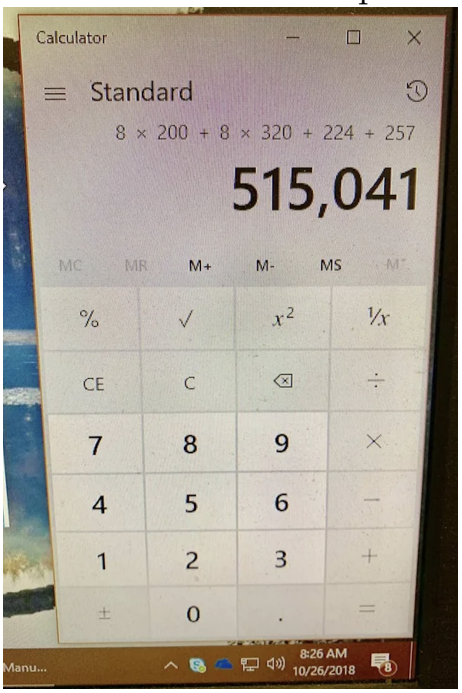
1.6 Reverse Polish Calculator

An application of the stack data structure is the RPN (reverse polish notation) calculator.

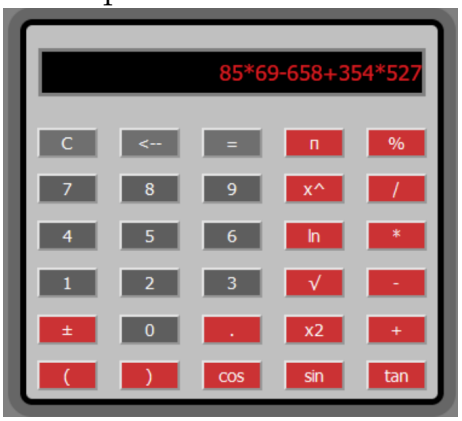
On a calculator how would you compute the following?

$$2 + 3 \times 4$$

It depends on whether the calculator implements algebraic order. Some calculators such as the original MS Windows calculator, would add $2 + 3$ when the user presses the $\times$ key.



Other calculators delay the addition operation until after 3×4 has been computed.



A scientific calculator will normally do the later; however, there are plenty of calculators which will interpret $2 + 3 \times 4$ as $(2 + 3) \times 4$ rather than as $2 + (3 \times 4)$. On a scientific calculator, if you want to compute $(2 + 3) \times 4$, you need `(` and `)` keys or at least some sort of way to *remember* values and use them later.

To compute the following, obeying the conventional alebraic order of operations, the computations need to be done in a particular order.

$$\underbrace{\underbrace{\underbrace{1}_{3}+\underbrace{2}_{7}}_{10}+\underbrace{\underbrace{3\times 4}_{11}-\underbrace{(5+6)}_{3}}_{33}\times(1+2)}_{-33}$$

To implement the algorithm to perform this computation, you'd need to attach *event-actions* to each key. The keys `0` ... `9` and `.` construct a number iteratively. The `+` and `-` keys finish any pending computation or computation pending inside an opening `(`. The $\times$ key pushes any pending addition or subtraction onto a stack, *etc.*. In fact, it is not immediately clear how to perform the computation correctly given the requirement that the correct result must be desplayed whenever it is possible, without knowing at which point in the future the user will press the `=` key.

Performing the computation above using RPN requires no parentheses to be used. However, such does require the operator (human being) to look-ahead in the mathematical expression.

To use an RPN calculator, the user presses digit keys including the decimal key and terminates a number by either pressing the `Enter` key or some mathematical operator. The `Enter` key pushes the number onto a stack; a mathematical operator key pushes the number onto the stack, and then peforms a mathematical operation by popping one or more arguments off the stack, internally computing and pushing the result onto the stack. Some operators such as $\sqrt{\cdot}$ and `cos` are unary operators, thus they pop, compute, and push. Other operators such as `+`, $\times$, and x^y are binary operators, thus thus they pop twice, compute, and push once.

To preform the following computation using RPN

$$1+2+3*4-(5+6)\times(1+2)\,,$$

the user must type the following keystrokes: `1`, `Enter`, `2`, `+`, `3`, `Enter`, `4`, $\times$, `5`, `Enter`, `6`, `+`, `1`, `Enter`, `2`, $\times$, `-`. Each time the `Enter` key or a mathematical operation key is pressed, the calculator does any necessary computation and re-displays the top of the stack.

User key presses	Displayed	Stack, Top...Bottom not displayed
1 Enter	1	1
2 +	3	3
3 Enter	3	3, 3
4 ×	12	12, 3
+	15	15
5 Enter	5	5, 15
6 +	11	11, 15
1 Enter	1	1, 11, 15
2 +	3	3, 11, 15
×	33	33, 15
-	-18	−18

Sometimes RPN allows certain optimizations in keystrokes.

If the user presses `Enter` without first entering digits, then the top of the stack is duplicated. This means the user can implement $17^2 + 17$ as `1`, `7`, `Enter`, `Enter`, `Enter`, ×, `+`.

User key presses	Displayed	Stack, Top...Bottom not displayed
1 7 Enter	17	17
Enter	17	17, 17
Enter	17	17, 17, 17
×	289	289, 17
+	306	306

1.6.1 Homework

There is a homework assignment `homework/rpn.py`; the test case is found in `tests/rpn_test.py`. The student should implement several unary and binary stack operations as Python functions.

The following are stack operations:

```
def dup(stack: Stack) -> Stack:

def swap(stack: Stack) -> Stack:

def pop(stack: Stack) -> Stack:
```

The following are mathematical operations:

```
def plus(stack: Stack) -> Stack:

def minus(stack: Stack) -> Stack:

def times(stack: Stack) -> Stack:

def fdivide(stack: Stack) -> Stack:

def idivide(stack: Stack) -> Stack:

def mod(stack: Stack) -> Stack:

def negate(stack: Stack) -> Stack:

def square(stack: Stack) -> Stack:

def cube(stack: Stack) -> Stack:

def reciprocal(stack: Stack) -> Stack:

def sqrt(stack: Stack) -> Stack:
```

In this file `Stack` is defined as `type Stack = Optional[tuple]`. The values of type `Stack` are those described in Section 1.4.

There is one additional function to complete: `eval_rpn_string`. This function is intended to be challenging.

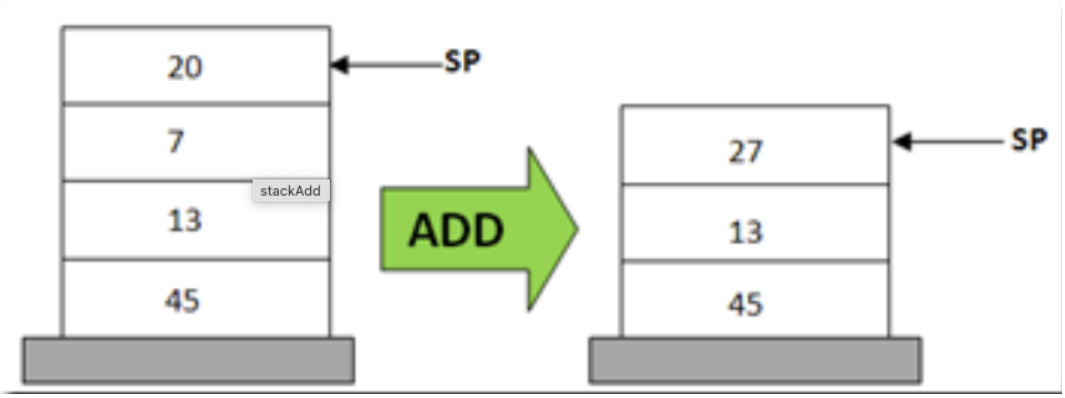
In the case of each function, the student is asked to complete the functionw with valid Python code which passes all the test cases provided in `tests/rpn_test.py`.

Chapter 2

Bartosz Stack Machine

Chapter Objectives

- Name the six VM state components (`code` , `stack` , `globals` , `pc` , `sp` , `fp`) and state the role of each.
- Read the `instructions` list: each triple is *(mnemonic, function, arity)*; the triple's index is the opcode.
- Trace the interpreter loop (`run`): fetch opcode, look up triple, call `ncode` arity-many times, dispatch to `instr_*` .
- Implement `push` , `pop` , and `ncode` correctly, including the `sp` edge cases.
- Explain the `CALL` / `RET` frame protocol: what is saved, in what order, and how `RET` restores the caller's state.
- Retrieve function arguments with `LDARG` *n* (arg 0 is the most recently pushed) and local variables with `LOAD` / `STORE` .
- Enforce 32-bit signed integer semantics via `justify()` .
- Assemble a textual program (mnemonics + labels) into an integer code array and label dictionary.



As a team project, you will implement a Virtual Machine in Python inspired by the **Simple Virtual Machine** described by Bartosz Sypytkowski. You may find a description of the original project at `https://www.bartoszsypytkowski.com/simple-virtual-machine/`. Bartosz's original project was conceived to be implemented in the C programming language, but ours will be in Python.

Bartosz Sypytkowski gratiously has granted permission to derive our work from his, and much of the documentation in this chapter comes verbatim from his original blog post, even if much of the documentation has been updated to more accurately describe our version of the VM.

There is already a fairly complete implementation in C published by Geoff Catlin at `https://github.com/gcatlin/simple-vm`. You are free to look at Catlin's implementation to gain inspiration. However, we are changing some of the specification, so be careful.

2.1 File Structure Overview

In your workarea you have several files, all with the prefix `bartosz`.

```
homework/bartosz_absval.py
homework/bartosz_arith.py
homework/bartosz_assembler.py
homework/bartosz_fibonacci.py
homework/bartosz_flow.py
homework/bartosz_memory.py
homework/bartosz_sanity_test.py
homework/bartosz_stack.py
homework/bartosz_vm.py
tests/bartosz_sample_test.py
```

You should search for all the lines containing the following comment, and add the appropriate Python code to implement the functions.

```
1    # CHALLENGE: student must complete the implementation.
```

The file `tests/bartosz_sample_test.py` contains a very small number of tests. You are expected to write your own tests by reading the descriptions in the function declaration stubs.

2.2 Student Group Assignment

The goal of this exercise is to complete the partial implementation of a modified version of the Bartosz Virtual machine. The project entails the following sub-goals.

1. Divide into teams of three or four persons each.

2. Each team should divide the work among the group members so that every person remains busy. However, each person should talk to other group members to keep them up to date, and discuss issues and decisions and questions.

3. Complete the `VM` class definition in file `bartosz_vm.py` by completing the methods: `push`, `pop`, `ncode`, and `run`. See Section 2.3.3

4. Each item in the `instructions` variable is a triple, mapping a mnemonic to a Python function and an arity (number of arguments that function needs to take). The function name is prefixed by `instr_`. Students must complete the implementation of all the `instr_` functions, which are grouped into different files according to semantics of the function in question.

5. The file `bartosz_sample_test.py` contains tests for some functions but not all. Students should finish all the tests, and may write additional tests as deemed necessary.

6. Each team can decide whether the same person writes the `instr_` function and its test, or whether one person writes the function and a different person writes the tests. Both approaches have advantages. If a different person writes the tests vs the function, then you have two people who must agree on what the function is supposed to do. This limits the consequences if one person misinterprets the docstring.

7. Implement `absolute_value` in Bartosz assembly language. Find the definition of `absval` in the file `bartosz_absval.py` and complete the code. This code should be able to compute the absolute value of any 32-bit integer except -2^{31} for which the absolute value is not representable because $|-2^{31}| = 2^{31}$, which is larger than the maximum 31-bit signed integer.

8. The team leader submits the proposed solution via Forge/Intra.

9. A team submits the solution files using Forge/Intra. A correct submission will include all the Python files `homework/bartosz_*.py` files as well as the `tests/bartosz_sample_test.py` file. A partial score *for the entire team* will be determined according to which percentage of the tests which pass. HOWEVER, students do not have access to the tests in this case.

10. The tests in `tests/bartosz_sample_test.py` will be run on a hidden/control (assumed correct) implementation of the Virtual Machine. If the student test detects a real error in the control solution, then bonus points will be awarded.

11. Certainly there are errors in the specification. When students find errors, you should discuss them as a team to make sure they are really errors, and report them back to the instructor.

2.3 Virtual Machine

This section describes many features necessary for the student to understand while implementing the VM.[1].

Let's start with describing what our virtual machine should consist of. We will create a stack-based virtual machine. What does it mean? There is no notion of registers, and all operation are performed through a virtual stack.

Our virtual machine will consist of:

- **Executable code array** — `vm.code` — this is a Python list of integers representing the program byte-code. Our byte-code contains opcodes (which are integers used to define instructions to be executed by an interpreter) and numbers used for things such as constant values, memory addresses or stack offsets. The value of `vm.pc` points to the index into `vm.code` where `vm.ncode()` will fetch the next byte code from.

- **Stack** — `vm.stack` — virtual stack will work as fragment of memory used for storing intermediate results and additional data of each instruction. When a procedure call will be invoked, it will also save a state of the program at moment when the procedure has been called, to restore back the state once program returns from that call.

- **Global data** — `vm.globals` — a list of globally available registers available to use as global variables. See the `GLOAD` and `GSTORE` instructions.

[1]This section is heavily based on Bartosz Sypytkowski's original blog post: `https://www.bartoszsypytkowski.com/simple-virtual-machine/` In many cases the text is exactly copied. In other cases it is modified to suite our purposes.

- **Local data** — unlike the stack, which uses LIFO (Last-In First-Out) operations, local data may be used as random access memory. It also may change its size according to program needs. We may use it to store local variables. See Section 2.3.7.
- **Program counter** — `vm.pc` — it contains an address of currently executed instruction. Programs will be executed by moving PC through code array (Python list), reading opcodes and executing them.
- **Stack pointer** — `vm.sp` — it contains information about number of elements stored on the stack. It always points on the top of the stack, or it will be -1 if the stack is empty.
- **Frame pointer** — `vm.fp` — since we need to know where local values start (local scope) (this includes function arguments), we need to know where our scope is located on the stack. All local variables are assigned relatively to FP position.

2.3.1 The `VM` Class and its Methods

The virtual machine is implemented as a Python class called `VM`. The template of the `VM` class is given here.

```
class VM:
    def __init__(self, code, pc, datasize, stack_size,
                 trace, single_step, ffi): ...

    def __call__(self, *args): ...

    def push(self, v): ...

    def pop(self): ...

    def ncode(self): ...

    def run(self, trace): ...
```

The `init` Magic Method

The `__init__` magic method serves to initialize an instance of the `VM` class. The method establishes several instance values onto a newly created instance of the class.

```
STACK_SIZE = 100
NUM_GLOBALS = 0


class VM:

    def __init__(self,
                 code: List[int],
                 pc: int = 0,
                 datasize: int = NUM_GLOBALS,
                 stack_size: int = STACK_SIZE,
                 trace: bool = False,
                 single_step: bool = False,
                 ffi: str = 'ffi'):
        self.globals = [0] * datasize
        self.labels, self.code = assemble(code)
        self.stack = [0] * stack_size
        self.stack_size = stack_size
        self.pc = pc
        self.sp = -1
        self.fp = -1
        self.trace = trace
        self.single_step = single_step
        self.ffi = ffi
```

The `__init__` magic method takes one mandatory argument `code` and several optional arguments.

- `self` — an uninitialized instance of the class `VM`, provided by the system.
- `code` — a list of either byte code, or assembly code to first be compiled and then interpreted.
- `pc` — (default to `0`) the initial index of the code array to be executed. This is the initial value of `vm.pc`.
- `datasize` — The number of global variables to allocate. The list `vm.globals` will be initialized to a list of `0` values indicated by this given size. Default is `NUM_GLOBALS`.

- `stack_size` — size of the stack to allocate. Default is `STACK_SIZE`. `vm.stack` will be initialized to a list of this size, filled with `0` values.
- `trace` — For debugging, this value instructs the interpreter to print diagnostic information before ech instruction is executed.
- `single_step` — An extension of `trace` which instrucs the interpreter to pause with a user prompt before each instruction is executed.
- `ffi` — See Section 2.4.6. This is the label which will designate where the entry point is if the `VM` object is called as a function.

The `call` Magic Method

The `__call__` magic method allows a `VM` instance to be called using Python function syntax. The given `args` are pushed onto the stack so that when the VM runs `args[0]` is accessible from `LDARG 0`, `args[1]` is accessible from `LDARG 1`, etc...

```
class VM:
    def __call__(self, *args):
        if self.ffi not in self.labels:
            raise RuntimeError(f'entry point {self.ffi} not in known
                                                labels')

        self.pc = self.labels[self.ffi]
        for s in reversed(args):
            self.push(s)

        self.run(trace=self.trace)
        return self.pop()
```

The `push` Method

The `push` method pushes the given argument onto the program stack, incrementing the stack pointer. First `vm.sp` is incremented, and THAT new index is used to reference into the stack. *E.g.*., if `vm.sp == 3`, then `vm.stack[4]` will become the value of `v` after `vm.sp` has become 4.

```
class VM:

    def push(self, v: int) -> None:
      ...
```

The `pop` Method

The `pop` method pops and returns the integer at top of stack, decrementing the stack pointer. The value of the stack at index `vm.sp` is returned, and `vm.sp` is decremented. *E.g.*., if `vm.sp` is 4, then `vm.sp` becomes 3, but `vm.stack[4]` is returned. If `vm.sp` is 0, then `vm.stack[0]` is returned, and `vm.sp` becomes -1. If `vm.sp` is already negative, an exception is thrown.

```
class VM:
    def pop(self) -> int:
      ...
```

The `ncode` Method

The `ncode` method reads the next integer from the code array, `vm.code`, returning the value at `vm.code[vm.pc]`, and increments `vm.pc`. *E.g.*., if `vm.pc` is 4, then `vm.code[4]` is returned, and `vm.pc` is incremented to 5. Note: `vm.pc` is an index into the list `vm.code`. If `vm.pc` points outside the list, then no error is thrown, rather the opcode of `HALT` is returned. This means users do not have to end their program with an explicit `HALT`, rather the program simply ends when it reaches the end of `vm.code`.

```
class VM:
    def ncode(self) -> int:
      ...
```

The `run` Method

See Section 2.3.3

2.3.2 Byte Code Instruction Set

At the beginning we start from defining a set of instructions to be handled by our language. For sake of simplicity we focus on basic integer operations, printing their result, variable creation and procedure-oriented calls.

The instructions are define by the global variable `instructions`.

```
instructions = [('NOP', instr_nop, 0),
                ('LABEL', None, 1),
                ('SYMBOL', None, 2),
                ('ADD_I32', instr_add_i32, 0),
                ('SUB_I32', instr_sub_i32, 0),
                ('MUL_I32', instr_mul_i32, 0),
                ('DIV_I32', instr_div_i32, 0),
                ('MOD_I32', instr_mod_i32, 0),
                ('BAND', instr_band, 0),
                ('BOR', instr_bor, 0),
                ('BXOR', instr_bxor, 0),
                ('BNOT', instr_bnot, 0),
                ('LAND', instr_land, 0),
                ('LOR', instr_lor, 0),
                ('LXOR', instr_lxor, 0),
                ('LNOT', instr_lnot, 0),
                ('LT_I32', instr_lt_i32, 0),
                ('EQ_I32', instr_eq_i32, 0),
                ('JMP', instr_jmp, 1),
                ('JMPT', instr_jmpt, 1),
                ('JMPF', instr_jmpf, 1),
                ('CONST_I32', instr_const_i32, 1),
                ('LOCALS', instr_locals, 1),
                ('LOAD', instr_load, 1),
                ('GLOAD', instr_gload, 1),
                ('STORE', instr_store, 1),
                ('GSTORE', instr_gstore, 1),
                ('PRINT', instr_print, 0),
                ('PRINTSTACK', instr_print_stack, 0),
                ('POP', instr_pop, 0),
                ('DUP', instr_dup, 0),
                ('HALT', None, 0),
                ('CALL', instr_call, 2),
                ('LDARG', instr_ldarg, 1),
                ('RET', instr_ret, 0)
                ]
```

Just like in standard assembly code, each instruction is represented by specific byte number. In more realistic virtual machine those numbers may have specific values (for example to perform bitwise masking).

The global variable `instructions` serves two purposes.

1. The existence of a triple in this list declares the existence and specification of an instruction in the instruction set.

2. The index of a triple in this list declares the byte code representing that instructions. *E.g.*, `ADD_I32` appears at index=3, therefore 3 is the opcode for `ADD_I32`.

Each triple in the `instructions` list specifies three things

1. The name of the instruction as a string, to be used in assembly programs.

2. The optional value of a Python function which implements the operation. The functions are named consistently with prefix `instr_`. If `None` is given as the function in this triple, that means the instruction is only an assembly-level instruction, not a real byte code. The assembler will take some action which the instruction is encountered, such as update the symbol table, but no byte-code is emitted into the code byte array.

3. The number of arguments of the instruction as an integer. This number indicates both the arity of the corresponding `instr_...` function, and also how many times `vm.ncode()` will be called to extract those arguments from the code array.

2.3.3 The Main Interpretor Loop

It's time to create our code execution loop. The `run` method in the `VM` class implements this loop. How does it work? The actual implementation in `bartosz_vm.py` contains additional features for debugging such as single stepping, which we omit here.

Its a simple `while True:` loop which

1. fetches an opcode,

2. looks up the opcode in `instructions`,

3. parses that triple to get the instruction function and arity,

4. calls `vm.ncode` arity-many times,

5. then calls the implementation function with the correct number of arguments,

The `run` method iterates through code array, `vm.code`, by successive calls to `vm.ncode()` either until the end is reached or until `HALT` is reached. Each instruction has an arity, so `vm.ncode` is called the correct number of times to collect the arguments, and then `run` must call the function implementing the instruction, and it must be called with the correct number of arguments. *E.g.*, if the instruction `LDARG` has been read (actually the integer corresponding to `LDARG`), then `LDARG` has `arity=1`, so `vm.ncode` is called once, and that value is passed as argument to the function `instr_ldarg`.

If `trace` has a true value, then print stack and print a disassembled rendition of the instruction at `vm.code[vm.pc]`.

TL;DR. Start executing the VM at the current value of `self.pc` and continue until `HALT` or the end of `self.code` is reached.

```
class VM:
  ..
  def run(self, trace=False) -> None:
    while True:
      opcode = self.ncode()
      ...
      mnemonic, instr, arity = instructions[opcode]

      if mnemonic == 'HALT':
        return

      # call ncode arity-many times, and build a list of return
      # values of ncode.  Then call the correct instruction
      # function with collected return values of ncode
      # as the function arguments.
         ...
```

2.3.4 Integer Operations

Now we can define some integer operations. While were defining plenty of them, here we show only one example for each group: constant initialization, arithmetic, and logic operations.

What is worth emphasizing, is that all value swaps are performed through stack (there are no registers), so each instruction takes its operands from top of the stack and also puts a computed result back onto the top of the stack. Also, since stacks are LIFO-type collections, all values taken from the stack are in reversed order — for and example look at the `ADD_I32` instruction below.

The `instr_add_i32` takes only `vm` as argument because the arity of the `ADD_I32` is 0—there are 0 additional arguments after `vm`. Note that `a` and `b` are in reverse order on the stack. The function assumes the stack ends in `[ ..... a, b]`., so that *vm.pop*() will return `b`, rather than `a`. The function, `instr_add_i32`, pops off `b` and then `a`, and pushes back the value of `a+b` as 32-bit signed integer. The result must be an integer $-2^{31} \leq n < 2^{31}$. This bounds enforcement should be done by the `vm.push()` method. The order in which `b` and then `a` are popped is not really important because the additions operator is commutative. However, the order is indeed important for other operations such as subtraction and division.

```
def instr_add_i32(vm: 'VM'):
    b = vm.pop()  # get second value from top of the stack ...
    a = vm.pop()  # ... then get first value from top of the stack ...
    vm.push(a + b)
```

The `instr_const_i32` takes two arguments, `vm`, and one additional called `value`, because the arity of the `CONST_I32` is 1—there is 1 additional argument after `vm`. This function should make a call to `vm.push(...)` pushing the value of `value` onto the stack.

```
def instr_const_i32(vm: 'VM', value):
    ...
```

The `instr_ls_i32` takes only `vm` as argument, because the arity of the `LT_I32` is 0—there are 0 additional arguments after `vm`.

```
def instr_lt_i32(vm: 'VM'):
    ...
```

This function assumes the stack ends in [..... a, b]. The function needs to pop off a and b in the correct order, and push back 0 (false) or 1 (true) depending on whether $a < b$. The assembler code may look something like this:

```
    CONST_I32 4
    CONST_I32 12
    LT_I32 # 4 < 12 ?
    JMPT 'some-label'  # the THEN part
    RET                # the ELSE part
```

Other instructions such as `SUB_I32`, `MUL_I32`, and `EQ_I32` wont be discussed here, they should be understandable from the example above, they should be easy to implement.

2.3.5 Jump Instructions

Next, we have to define jump instructions in our code. They are basically equivalent of `goto` keyword and may be used to implement statements such as loops and if/then/else conditions.

The `instr_jmp` function has one argument after `vm` which contains the address to jump to. This address is the index in the code vector. The program counter must be set to that explicit value.

Set the program counter to the specified, explicit address, *i.e.* unconditionally jump with program counter.

```
def instr_jmp(vm: 'VM', addr):
    vm.pc = addr
```

Again, `JMPF`, conditional jump if false, and `JMPT`, conditional jump if true, are implemented similarly to `instr_jmp` except that a value must be popped from he stack, and the program counter set only if the popped value is the correct Boolean value. 0 is considered false, every other value is considered true.

2.3.6 Global Memory

Now, when all basic constructs are defined, we shall provide a way to define a variables in our program. This section covers problem of globally-scoped ones, while the next one will show how to implement locally-scoped variables.

Unlike stack, a local data is a random access memory of fixed size, so all in/out operations performed on it indexes into this globally available

array (Python list). Therefore, in our example all load/store opcodes should be followed by number being an index to the local data.

These functions are fully implemented for the student, so there should be nothing remaining to do.

The function, `instr_gload` implements the `GLOAD` instruction, for pushing the value of a global variable onto the stack. Analogously, the `GSTORE`, instruction, implemented by the function `instr_gstore` pops a value off the stack and stores that value into a global variable. In each case the global variable is simply indicated by the given index.

```
def instr_gload(vm: 'VM', addr):
    v = vm.globals[addr]
    vm.push(v)

def instr_gstore(vm: 'VM', addr):
    v = vm.pop()
    vm.globals[addr] = v
```

2.3.7 Local Memory

This is similar to previously defined `GLOAD` and `GSTORE` instructions. However, there is a one significant difference — instead of taking an locals address directly from provided value, we compute it relatively to current frame pointer.

The `LOCALS` instruction, implemented by `instr_locals`, reserves the given number of local variables, on the stack but starting at the frame pointer. The `LOCALS` instruction should normally only be executed immediately on function entry. If a value of `vm.fp` differs from `vm.sp` an exception will be raised, and VM execution will halt.

The function pushes 0 or more 0's onto the stack. These zeros effectively reserve space for `local_count`-many local variables.

```
def instr_locals(vm: 'VM', local_count):
    assert vm.sp == vm.fp, "cannot execute LOCALS instruction after
                                    stack has been modified"
    for i in range(local_count):
        vm.push(0)
```

After a `LOCALS n` instructions has been executed, there will be the

specified number of reserved local variables on the stack, numbered from 0 to $n-1$. These can be accessed (read) using `LOAD` instruction and may be modified using the `STORE` instruction. Successive calls to push will not conflict with these local variables as the `LOCALS` instruction updates the stack pointer accordingly.

The `instr_load` function implements the `LOAD` instruction, accepting a single argument (in addition to `vm`). The argument specifies the index of the local variable in question. The function must push the indicated local variable onto the stack. Local variable i may be found by `vm.fp + i + 1`.

```
def instr_load(vm: 'VM', offset):
  ...
```

The `instr_store` function implements the `STORE` instruction, accepting a single argument (in addition to `vm`). The argument specifies the index of the local variable in question. The function pops a value off the stack and stores it into the local variable at the indicated index. This index can be calculated, for variable i, at location `vm.fp + i + 1`.

```
def instr_store(vm: 'VM', offset):
    ...
```

It is the responsibility of the programmer to assure that programs never access variables that have not been allocated. For example the following code would give unpredictable results, because 3 local variables are *reserved*, indexed, $0 \ldots 2$. However, the program immediately attempts to access the variable at index 5, which has not been reserved.

```
  LOCALS 3
  CONST_I32 42
  STORE 5
```

2.3.8 Procedure Calls

All of previous instructions were fairly simple to execute. Now its time to do something slightly more complex — function/procedure calls. Before we begin, let's describe how they work. A procedure call is launched by the `CALL` instruction, implemented by the function `instr_call`.

The job of the `instr_call` function is to manipulate the state of the VM to effectuate the requested call, but also to leave evidence which `RET` can later use to effectuate the `return`. We have to first save the current state of the program at the moment before the call will be performed — it will let us freely alter program state in scope of the function. To do so, we have to save three values:

1. **Address of the caller instruction** — this is a current value of program counter and its need to be stored, so that VM will know, where program should return from finished procedure call.

2. **Frame pointer** — this is necessary, since we want to be able to use locally scoped variables when we enter into new procedure (TL;DR it gives us a function scope feature).

3. **Number of function/procedure arguments** — before call will be performed, all necessary arguments will have been put on top of the stack. However returning from function should dispose all of the provided arguments. So that we could retrieve back state of the stack from before the function call. If we want to know how many arguments from the stack should be thrown away, firstly we have to store that count — it will be also put on top of the stack.

The `RET` instruction works in reverse order:

1. It takes value from top of the stack as function return value,

2. rolls back the frame pointers,

3. moves the program counter to the caller instruction,

4. disposes all of function call arguments from the stack and replaces all of these with return value.

Because we have stored all of those values on top of the stack, were able to store next consecutive calls in the same manner. It means that, were able to nest our procedure calls. However, since each call stores some portion of data on the stack, at some point we may run out of memory, which cause stack overflow error.

The `instr_call` function must implement the following function call protocol:

1. We expect that stack has argc-many values having been pushed by the caller.
2. We expect all args to be on the stack.
3. Save the three values: `vm.argc`, `vm.fp`, and `vm.pc` onto the stack.
4. Update `vm.fp` to `vm.sp`
5. Update `vm.pc` to the address of the function being called.

```
def instr_call(vm: 'VM', addr, argc):
    ...
```

The `instr_ret` function must implement the following function return protocol, which is pretty much the inverse operation of the function call protocol. Restore the stack to *almost* its state before the function was called.

1. The top of the stack is the value to be returned, so *stash* that value aside, then it, the local variable (which we actually do not care about) and the three values which `CALL` pushed,
2. Restore `vm.sp`, `vm.pc`, and `vm.fp` to what their values were before `CALL`.
3. Subtract `argc` from `vm.sp`.
4. Finally, push the return value (which was *stashed*) back onto the stack.

```
def instr_ret(vm: 'VM'):
    ...
```

The instruction `LDARG`, implemented by the function `instr_ldarg` allows a function written in assembly language to access variables that are passed to a function call. Recall the `CALL, addr, argc,` specifies how many values are promised to have been pushed onto the stack to serve as function parameters for the function at the specified `addr`. Within

that function, subsequent to the `CALL` instruction having been executed but prior to the `RET` instruction's execution, these values are available as function arguments, and accessible with the `LDARG` instruction.

The function (via `CALL`) has been called with 0 or more arguments. The `instr_ldarg` must Get the indicated argument and push it onto the stack. Arg 0 is the argument most recently pushed. Arg 1 is the argument pushed prior to that. The index of the specified argument is relative to `vm.fp`, but skipping the 3 values which `CALL` pushed, i.e. `argc`, `vm.fp`, and `vm.pc`.

```
def instr_ldarg(vm: 'VM', arg):
    ...
```

Here is an example of how to call a function in the assembly language.

```
CONST_I32 42        # arg 2
CONST_I32 102       # arg 1
CONST_I32 53        # arg 0
CALL 'xyzzy' 3
PRINT               # pop and print 4337
HALT
LABEL 'xyzzy'
LDARG 0             # push 53 onto the
LDARG 1             # push 102 onto the
LDARG 2             # push 42 onto the
MUL_I32
ADD_I32
RET                 # returns 42*102 + 53 = 4337
```

2.4 Important Differences

While you will certainly get great inspiration from the blog post `https://www.bartoszsypytkowski.com/simple-virtual-machine/` as well as the sample implementation `https://github.com/gcatlin/simple-vm`, there are several important differences to understand in our implementation.

2.4.1 Additional Mnemonics

The following mnemonics are not found in the reference implementation. Some of these already have full and finished implementations. Others must be completed by the student team.

- `NOP`: Empty Operation
- `LABEL`: Associate symbolic name with current code location (array index)
- `SYMBOL`: Associate symbolic name with another 32-bit integer.
- `DIV_I32`: 32-bit quotient of integer division
- `MOD_I32`: 32-bit remainder of integer division
- `BAND`: bitwise AND
- `BOR`: bitwise OR
- `BXOR`: bitwise XOR
- `BNOT`: bitwise NOT
- `LAND`: logical AND
- `LOR`: logical OR
- `LXOR`: logical XOR
- `LNOT`: binary NOT
- `LOCALS`: allocate given number of local variables on the stack, all initialized to 0. These can be accessed with `LOAD` and `STORE`.
- `PRINTSTACK`: print stack for debugging
- `DUP`: duplicate top of stack
- `LDARG`: allocate (read) a function argument. Function arguments work as in the following example.

```
code = ['CONST_I32', 41,  # arg 2
        'CONST_I32', 42,  # arg 1
        'CONST_I32', 43,  # arg 0
        'CALL', 'test-3-arg', 3,
        # print value returned from test-3-arg,
        'PRINT',          #  ie. print 126
        'HALT',
        'LABEL', 'test-3-arg',
        # now return arg-0 = 43,
        #            arg-1 = 42,
        #        and arg-2 = 41
        'LDARG', 1,  # push 42 onto the stack
        'LDARG', 2,  # push 41 onto the stack
        'LDARG', 0,  # push 43 onto the stack
        'ADD_I32',   # [42 41 43] --> [42 84]
        'ADD_I32',   # [42 84] --> [126]
        'RET'        # return 126
]
```

Several mnemonics have different semantics.

- `LOAD` n: access (read from and push onto the stack) the n'th (zero-based) local variable which was allocated by `LOCALS`.
- `STORE` n: modify the n'th (zero-based) local variable which was allocated by `LOCALS`. A value is popped of the stack and stored at local location n.
- `GLOAD`: Access (read from and push onto the stack) a block of global memory stored in `vm.globals[]`. The size of this array is determined when the `VM` is initialized, *i.e.*, during the call to `VM(...)`.
- `GSTORE` n: modify the n'th (zero-based) global memory location stored in `vm.globals[]`. A value is popped off the stack and stored into this array at the given index `n`.

2.4.2 32-bit Arithmetic

In this VM, we will enforce 32-bit signed arithmetic. Integers are in the range $-2^{31} \leq n < 2^{31}$. *I.e.*, there are 2^{32} distinct integers, 2^{31} are negative, $2^{31} - 1$ are positive.

2.4.3 Program Listing

A program listing will be a valid Python list such as

```
    code = ['LABEL', 'begin',
            'SYMBOL', 'val', 42,
            'CONST_I32', 'val',
            'CALL', 'sample-fun', 1,
            'HALT',
            'LABEL', 'sample-fun',
            'LDARG', 0,
            'DUP',
            'MUL_I32',
            'CONST_I32', 2,
            'MUL_I32',
            'RET'
           ]
```

Such an argument is given as first argument to `VM(...)`.

The assembler, the `assemble` function, is already fully implemented.

Internally, the `assemble` function checks this code list for errors, converts it to a list of pure 32-bit integers, and build a label dictionary. For the example above, the `assemble` returns a label dictionary such as

```
   {'begin': 0,
    'val': 42,
    'sample-fun': 6
   }
```

and a code array such as

```
   [19, 42, 30, 6, 1, 29, 31, 0, 28, 5, 19, 2, 5, 32]
```

The specific values in the integer form (the assembled form) of the code array depend on the order the opcodes are listed in the variable `instructions`.

```
# each triple indicates a human-readable mnemonic, function object,
                                    and arity
instructions = [('NOP', instr_nop, 0),  # Empty Operation
                ('LABEL', None, 1),
                ('SYMBOL', None, 2),  # associate a symbol with value
                ('ADD_I32', instr_add_i32, 0),  # int add
                ('SUB_I32', instr_sub_i32, 0),  # int sub
                ('MUL_I32', instr_mul_i32, 0),  # int mul

                # lines skipped ...

                ('CALL', instr_call, 2),  # call procedure
                ('LDARG', instr_ldarg, 1),
                ('RET', instr_ret, 0)  # return from procedure
                ]
```

The `assemble` method converts the code array which contains strings such as `'LABEL'`, `'ADD_I32'`, *etc.* to a array of pure 32-bit signed integers.

2.4.4 Automatic Arity Handling

Bartosz's C implementation of the VM contains a C `switch` statement to dispatch an op code to the correct implementation code. Each `case` in this `switch` is responsible for calling `NCODE` the correct number of types to advance the program counter `vm->pc` to the next opcode for the next instruction. In the following example `HALT` and `LT_I32` call `NCODE` zero times, `CONST_I32` calls `NCODE` once, and `CALL` calls `NCODE` twice.

```
void run(VM* vm){
  do{
    int opcode = NCODE(vm);
    int v, addr, offset, a, b, argc, rval;

    switch (opcode) {
      case HALT: return;
      ...
      case CONST_I32:
        v = NCODE(vm);
        ...
        break;

      ...

      case CALL:
        addr = NCODE(vm);
        argc = NCODE(vm);
        ...
        break;

      case LT_I32:
        ...
        break;

       ...
      default:
        break;

    } while(1);
}
```

Whereas it is the responsible of the code handling each and every opcode (in Bartosz's implementation) to call `NCODE` the correct number of times, in our representation, the `run` function uses the `instructions` array to obtain the arity and make the appropriate number of calls to `ncode` and then call the appropriate dispatch function `instr_const_i32`, `instr_call`, `instr_lt_i32`, *etc.*, with the correct number of arguments.

This means the dispatch functions such as `instr_const_i32`, `instr_call`, `instr_lt_i32`, *etc.*, never need to call `ncode`, but rather already have corresponding function arguments provided.

```
def instr_const_i32(vm: 'VM', value):
  ...

def instr_call(vm: 'VM', addr, argc):
  ...

def instr_lt_i32(vm: 'VM'):
  ...
```

2.4.5 Debugging features

`emulate`: This function is useful for testing. The function runs the given code by

- First assemble it using the `assemble` function.
- If the `stack` argument is given, then these values are pushed onto the stack before the execution begins. Note the values are pushed in order so that `stack[0]` can be accessed with `LOAD 0`, `stack[1]` can be accessed with `LOAD 1`, `stack[2]` can be accessed with `LOAD 2`, *etc.* Note that when `emulate` is being used with the `ffi` interface, the literal order of these arguments is the same as explained here. This is explained in detail in Section 2.4.6.
- Then, execute either starting at index 0, or at the specified `entry` label.
- `emulate` returns the possibly empty stack (as a list) as it remains after the program finishes executing.

The return value of `emulate` can be used for writing test cases such as the following.

```
def test_add_2(self):
  for a in range(-num_repetitions, num_repetitions):
      for b in range(-num_repetitions, num_repetitions):
          self.assertEqual(emulate(['CONST_I32', a,
                                    'CONST_I32', b,
                                    'ADD_I32']),
                           [a + b])
```

`ffi`: See Section 2.4.6

2.4.6 Foreign Function Interface

The `VM` class implements the `__call__` method which means the object may be called as a function.

```
vm = VM([ ... 'LABEL', 'ffi', ...])
vm(...)
```

The call to `vm` must take exactly as many arguments as indicated in the assembly code. In the example below `'CALL', 'test', 3` indicates that `vm` needs three arguments. Those arguments are pushed onto the stack before the `run` method is executes starting at `'LABEL', 'ffi'`.

Moreover, the order they appear in the call-site `vm(10, 35, 99)` corresponds exactly to the order they may be retrieved within the assembly code with `'LDARG', 0`, `'LDARG', 1`, and `'LDARG', 2`. After the program halts, whichever value is left atop the stack becomes the return value of `vm`. Other elements on the stack are ignored.

```
code = ['LABEL', 'test',
        # compute arg0 * arg1 + arg2
        'LDARG', 0,
        'LDARG', 1,
        'MUL_I32',
        'LDARG', 2,
        'ADD_I32',
        'RET',

        'LABEL', 'ffi',
        # call test, but with arguments of vm(...)
        # having been pushed on the stack
        'CALL', 'test', 3,
        'HALT'
]

vm = VM(code)

# call 'test' with LDARG 0 = 10
#                  LDARG 1 = 35
#                  LDARG 2 = 99
vm(10, 35, 99)  # compute (10 * 35) + 99
```

2.5 Example Program

To illustrate the differences between the C-version of the Bartosz Virtual Machine and our Python implementation, we show a program directly from the Bartosz Virtual Machine blog post, and the Python version of the same function.

```
const int fib = 0;  // address of the fibonacci procedure
int program[] = {
    // int fib(n) {
    //     if(n == 0) return 0;
    LOAD, -3,
    CONST_I32, 0,
    EQ_I32,
    JMPF, 10,
    CONST_I32, 0,
    RET,
    //     if(n < 3) return 1;
    LOAD, -3,
    CONST_I32, 3,
    LT_I32,
    JMPF, 20,
    CONST_I32, 1,
    RET,
    //     else return fib(n-1) + fib(n-2);
    LOAD, -3,
    CONST_I32, 1,
    SUB_I32,
    CALL, fib, 1,
    LOAD, -3,
    CONST_I32, 2,
    SUB_I32,
    CALL, fib, 1,
    ADD_I32,
    RET,
    // entry point - main function
    CONST_I32, 6,
    CALL, fib, 1
    PRINT,
    HALT
};

// initialize virtual machine
VM* vm = newVM(program,   // program to execute
                  38,     // start address of main function
                  0);     // no locals
run(vm);
```

Now the Python version of the same function.

```
fibonacci = [
    # int fib(n) {
    #     if(n == 0) return 0;
    'LABEL', 'fib',
    'LDARG', 0,
    'CONST_I32', 0,
    'EQ_I32',
    'JMPF', 'if(n < 3)',
    'CONST_I32', 0,
    'RET',

    #     if(n < 3) return 1;
    'LABEL', 'if(n < 3)',
    'LDARG', 0,
    'CONST_I32', 3,
    'LT_I32',
    'JMPF', 'N >= 3',
    'CONST_I32', 1,
    'RET',

    #     else return fib(n-1) + fib(n-2);
    'LABEL', 'N >= 3',
    'LDARG', 0,
    'CONST_I32', 1,
    'SUB_I32',
    'CALL', 'fib', 1,
    'LDARG', 0,
    'CONST_I32', 2,
    'SUB_I32',
    'CALL', 'fib', 1,
    'ADD_I32',
    'RET',

    'LABEL', 'ffi',
    'CALL', 'fib', 1,
    'HALT'
]

  vm = VM(fibonacci)
  vm(8) # compute 8th fibonacci number
```

Chapter 3

Binary Trees

Chapter Objectives

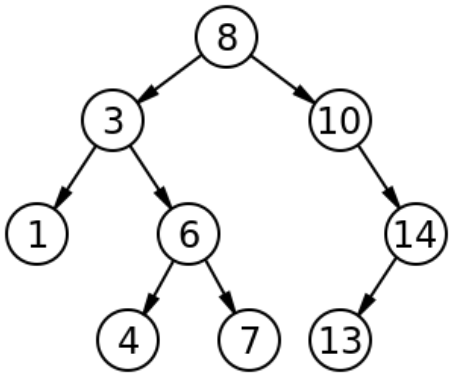
- Represent a BST as nested Python 3-tuples `(key, left, right)` and implement recursive `search`, `insert`, `delete`, `serialize`, and `rebalance`.
- Explain the BST invariant and why `serialize` produces a sorted list without calling `sort`.
- Represent a heap as a 1-based array and derive the parent/child index formulas for both zero-based and one-based encodings.
- Implement `sift_up` and `sift_down`, and use them to build `insert`, `extract`, and `heapify`.
- Prove that `heapify` runs in $O(n)$ time using the closed-form bound $\sum_{k=0}^{\infty} k(1/2)^k = 2$.
- Explain how a Huffman tree induces a prefix code, and trace the greedy algorithm that builds the optimal tree using a min-heap.
- Implement `build_tree`, `tree_to_encoding`, `encode_message`, and `decode_message` for Huffman coding.

A *binary tree* is a data structure made of leaves and internal nodes, in which no internal node has more than two children nodes. In some

special cases, a binary tree may be required to be more restrictive in that every internal node have exactly two child nodes.

3.1 Binary Search Trees

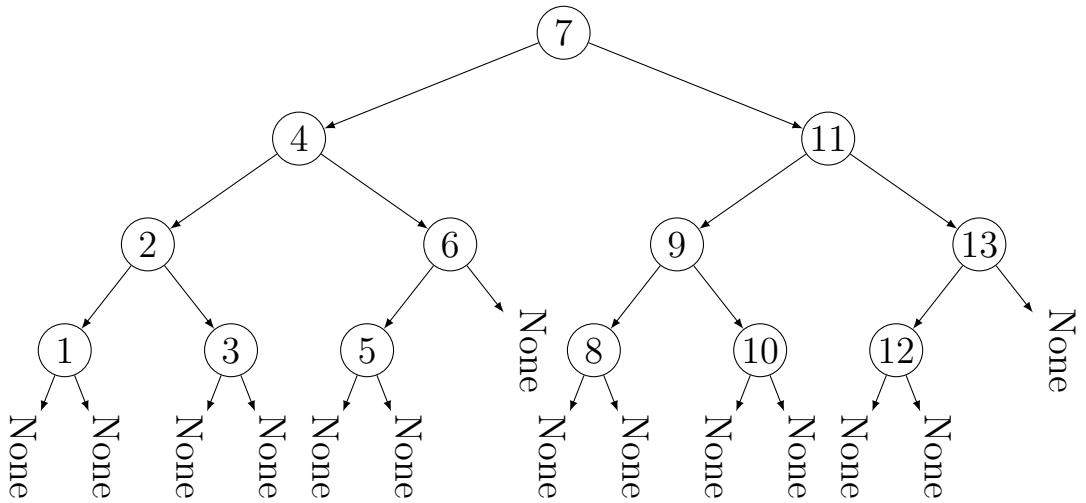
A BST (binary search tree) is a binary tree which associates each internal node with a key, where the keys come from an ordered set. In a BST each internal node has the property that its key is greater than (or equal to) to all the keys of the nodes reachable from its left-hand child, and simultaneously less than (or equal to) than the keys of all the nodes reachable from the right-hand child.



If data is stored in a balanced BST, it can be found quickly. When searching for a given value in a BST, we look at the key at the node, either we find the value equal to the value being sought, and if not, we immediately know whether to search the right or left child. *I.e.*, we immediately eliminate half the search space. This elimination of half the search space applies again when searching the child node. Thus with each level walked, another half of the search space is eliminated. We can only repeat this process $\log_2(n)$ times. Thus no search needs more than $\log_2(n)$ steps (assuming the tree is balanced).

In this chapter we will only deal with immutable BSTs, as the associated algorithms are generally easier and cleaner. Moreover, the computational complexity of mutable vs immutable BSTs can be made identical.

Furthermore, we will make a simplifying assumption in our Python implementation of BST, that all internal nodes have exactly two children, and all leaf nodes are identically `None`.



We will represent an internal node as a python 3-tuple where the first element is the key (1 through 13 in this case), followed by two nodes (denoting the left and right child in that order), each of which is either `None` or another such 3-tuple.

To represent the tree shown above the following Python code suffices.

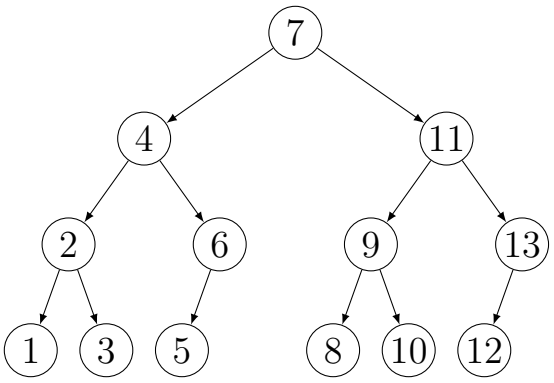
```
# internal nodes of level 3
node1 = (1, None, None)
node3 = (3, None, None)
node5 = (5, None, None)
node8 = (8, None, None)
node10 = (10, None, None)
node12 = (12, None, None)

# internal nodes of level 2
node2 = (2, node1, node3)
node6 = (6, node5, None)
node9 = (9, node8, node10)
node13 = (13, node12, None)

# nodes of level 1
node4 = (4, node2, node6)
node11 = (11, node9, node13)

# top node of level 0
node7 = (7, node4, node11)
```

We will also omit drawing of the leaf nodes as they are always `None`.



3.1.1 Search

The following code searches for a given key in the given tree. If such is found, the node is returned, *i.e.*, a triple. If not, `None` is returned.

```
def bst_search(node, key):
  if node is None:
    return None
  k, left, right = node
  if k == key:
    return node
  elif key < k:
    return bst_search(left, key)
  else:
    return bst_search(right, key)
```

3.1.2 Insert

To insert into a BST, there are two possibilities, depending on what you want to do in the case of duplicate elements. *I.e.*, if you attempt to insert a key which is already in the BST, would you like to insert it again, or would you like to simply do nothing?

The following will insert, ignoring duplicates

```
def bst_insert(node, key):
  if node is None:
    return (key, None, None)

  k, left, right = node
  if k == key:
    return node
  elif key < k:
    return (k, bst_insert(left, key), right)
  else:
    return (k, left, bst_insert(right, key))
```

The following will insert duplicates on the left.

```
def bst_insert(node, key):
  if node is None:
    return (key, None, None)

  k, left, right = node
  if key <= k:
    return (k, bst_insert(left, key), right)
  else:
    return (k, left, bst_insert(right, key))
```

3.1.3 Sorting with a BST

To sort a list using a BST, we simply iteratively insert into a BST, starting with an empty tree, when serialize the tree. First let's look at serialize. A serialized version of a BST is a list of the keys, respecting the order imposed by the tree.

```
def bst_serialize(node):
  if node is None:
    return []

  k, left, right = node
  return bst_serialize(left) + [k] + bst_serialize(right)
```

Now, we can sort using `bst_serialize`

```
def bst_sort(data):
  from functools import reduce

  return bst_serialize(reduce(bst_insert, data, None))
```

This can be done without the `reduce` function.

```
def bst_sort(data):
  bst = None
  for item in data:
    bst = bst_insert(bst, item)

  return bst_serialize(bst)
```

3.1.4 Rebalancing a BST

Searching in a BST is fastest if it is balanced. However, it can only be perfectly balanced if the number of internal nodes is one less than a power of two: $2^n - 1$ for some $n \in \mathbb{N}$. Otherwise, it can be approximately balanced. We look here at one algorithm for rebalanced a BST.

The idea is to:

1. Serialize the tree into a list, using the `serialize` you've already written.
2. Split this list into three parts, (1) the center element, called the *pivot element*; (2) everything to the left, called the *prefix*; and (3) everything to the right, called the *suffix*.
3. Build a new BST node containing the pivot element as element, and build left and right child trees out of the prefix and suffix respectively. The child trees can be built recursively. (4) You can implement the termination conditions by simply detecting the empty list and returning `None`. However, you can optimize the termination condition by detecting the singleton, or doubleton lists.

```
def build_ordered(items):
   ...

def rebalance(tree):
   return build_ordered(serialize(tree))
```

3.1.5 Removing from a BST

The BST remove function is not as elegant as the other functions because there are several cases to consider. Removing a leaf node is simple, but

if the node we with to remove is not a leaf node, then we must remove a node, and re-attach the stranded children. Additionally, there may be two children, or perhaps only a left-child, or perhaps only a right-child. All these cases must be considered.

- If we are removing a node from an empty tree, then just return the empty tree.
- If the item is on the left-left side, then construct a new node `(value, RECUR, right)`,
- If the item is on the right-hand side, then construct a new node `(value, left, RECUR)`.
- If both children are empty, then return an empty tree.
- If exactly one child is empty, return the other child.
- If both children non-empty trees ....

The most difficult case is if we are removing a node which has two children. In this case we effectively want to replace the node with another tree which has one fewer nodes. So we either replace the node with the maximum element from the left child, or we replace the node with the minimum element from the right child.

Since all the items in the left child are less than (or equal) to the element being removed, then we find the maximum such element. This maximum element is the immediate predecessor of the key begin removed. We can form a new node having the predecessor as key. If we simply build a node `(predecessor, left, right)` then we will have duplicated the predecessor, so we need to remove the predecessor from the left child. *I.e.*, we instead return the node `(predecessor, bst_remove(left, predecessor), right)`.

```
def bst_remove(node, item):
    if node is None:
        return None
    value, left, right = node
    if item > value:
        return (value, left, bst_remove(right, item))
    elif item < value:
        return (value, bst_remove(left, item), right)

    # if both children are None, return None
    if left is None and right is None:
        return None
    # if the node has exactly one child, i.e.
    # if one child is None, return the other child
    elif left is None:
        return right
    elif right is None:
        return left

    # else we are removing an internal node with two children,
    # we must remove the node, and stitch the tree back together.
    else:
        pred = max_tree(left)
        return pred, bst_remove(left, pred), right
```

3.2 Priority Queue

A *priority queue* is a data type which implements a collection. Each item in the collection is tagged with a priority. A critical feature of a priority queue is that it is efficient to find and extract (remove) the element with highest priority, and that that operation can be performed repeatedly to extract successive elements efficiently. A caveat is that to remove the element of second highest priority, we must first remove the element of highest priority.

In practice there are two types of priority queues: the min-queue and the max-queue. A min-queue makes the minimum element easily accessible, and a max-queue makes the maximum element easily accessible.

A priority queue provides the following interface:

min-queue	**max-queue**
`insert(element,priority)`	`insert(element,priority)`
`minimum()`	`maximum()`
`extract-min()`	`extract-max()`

Sometimes an additional operation is implemented: `increase-key(element,new-priority)` (to increase the priority of an element in a max-queue) or `decrease-key(element,new-priority)` (to decrease the priority of an element in a min-queue). For example the Dijkstra shortest path algorithm makes use of the `decrease-key(element,new-priority)` operation.

There are many different implementations of priority queue. A very popular one is the `min-heap` / `max-heap`. A heap can be though of abstractly as a binary tree, but is usually encoded as an array as shown in Figure 3.1.

In the array encoding, the indexes of the children nodes are computed as a function of the index of the parent node by either a zero-based or one-based encoding. This encoding assumes the array is filled from left to right; *i.e.*, there can be no gaps between the leaf nodes. In Figure 3.1 we see there is no gap between the leaves: 3, 25, 1, 2, and 7. If we wish to add another element, the next available space is to the right of 7, *i.e.*, as a left-child of node 3.

In a zero-based encoding, the left and right children of the node at index i, are stored at indices $2i + 1$ and $2i + 2$, respectively. Conversely,

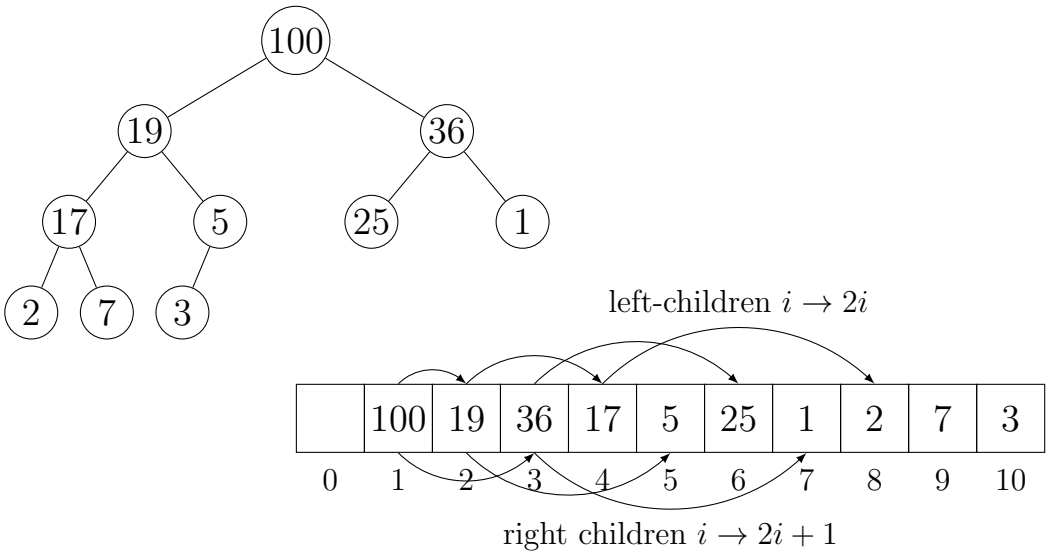


Figure 3.1: Array encoding of heap

the parent index (the index of the parent of a node) can be computed given the index of the child node as $\left\lfloor \frac{i-1}{2} \right\rfloor$. This formula works for both the left and right child.

In a one-based encoding, the left and right children of the node at index i, are stored at indices $2i$ and $2i + 1$, respectively. Conversely, the parent index (the index of the parent of a node) can be computed given the index of the child node as $\left\lfloor \frac{i}{2} \right\rfloor$. This formula works for both the left and right child.

parent	**one-based left**	**one-based right**	**zero-based left**	**zero-based right**
i	$2i$	$2i + 1$	$2i + 1$	$2i + 2$
0			1	2
1	2	3	3	4
2	4	5	5	6
3	6	7	7	8
4	8	9	9	10
5	10	11	11	12
⋮	⋮	⋮	⋮	⋮

In the one-based encoding, the 0-index is unused, or can be considered a free location to use for any kind of bookkeeping task, *e.g.*, to hold the queue length in programming languages where an array does not

implicitly know its size.

3.2.1 Implementing the Heap API

There are four main functions to implement in the heap API, `create_heap`, `insert`, `extract`, and `heapify`. These functions rely on some low level functions: `sift_up` and `sift_down`.

- `create_heap`: Creates an empty heap, specifying whether this is a min-heap or max-heap.
- `insert`: Insert a given element into a valid heap.
- `extract`: Remove and return the element at the top of the heap, this is either the minimum or maximum element depending on whether the heap is a min-heap or a max-heap.
- `heapify`: Given a list (in unspecified order), create a heap (by calling `create_heap`), and populate the heap with all the items from the given list. However, do not repeatedly call `insert`, as this would be slow. Instead, use repeated calls to `sift_down` to rearrange the elements to form a valid help.

- `sift_up`: Walk the lineage from a given child up to the top of the heap (short-circuiting if possible). As the walk is performed, the parent and child are swapped if they are in the wrong order. If a swap is performed, keep walking up. If a swap is not necessary stop walking.
- `sift_down`: Walk down from the given parent either the left or right child. As the walk is performed, examine the three elements, parent, left-child, right-child. If the parent-child relationships are correct, then stop. Otherwise one (and only one) of the children should be swapped with the parent to enforce the correct ordering. Continue the walk by walking into the effected child. There are several special cases: zero children, only left child, only right child, two children.

3.2.2 The `create_heap` function

In some implementations of heap, the `create_heap` has nothing to do. Some implementations just use an array to implement the heap, and use some local variable to remember the size. However, there are at least two pieces of information that need to be maintained with the heap, (1) the size (so we can know where the end of the heap is) and (2) the invariant, so we can know whether this is a min-heap or max-heap. The `create_heap` function gives us a place to encode this information somewhere.

In the implementation you have as a homework problem (Section 3.2.10) you will encode this information as a dictionary stored at index zero of the array (a Python `list`).

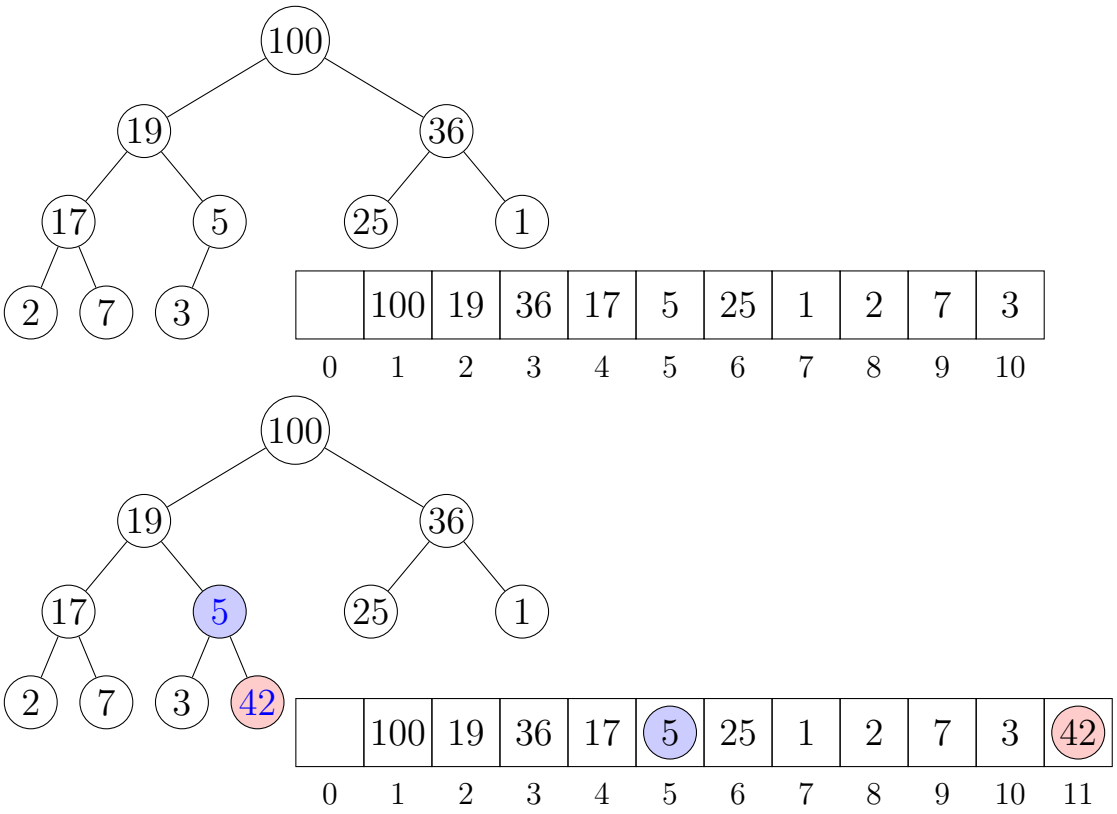


Figure 3.2: [Top] Heap before inserting. [Bottom] heap invariant violated after inserting 42; `sift_up` needs to swap 5 and 42.

3.2.3 The `insert` function

If we want to insert a new element into the heap it must be placed at the end of the array. But appending such an element likely invalidates the heap invariant. The parent of the newly added element may not be greater (in the case of max-heap) than the newly added child. To fix this, we simply call `sift_up` on the newly added child.

As an illustration, suppose we begin with the heap shown in Figure 3.2 [Top]. If we wish to insert, we will need to create a right child of the node 5. If, by chance, we insert some value which is less than 5, then the heap remains correct, and no sifting will be necessary. So let's insert 42. Immediately after insertion, we have an invalid heap as shown in Figure 3.2 [Bottom], because $5 \geq 42$ is false. A call to `sift_up` will fix this. See Section 3.2.6.

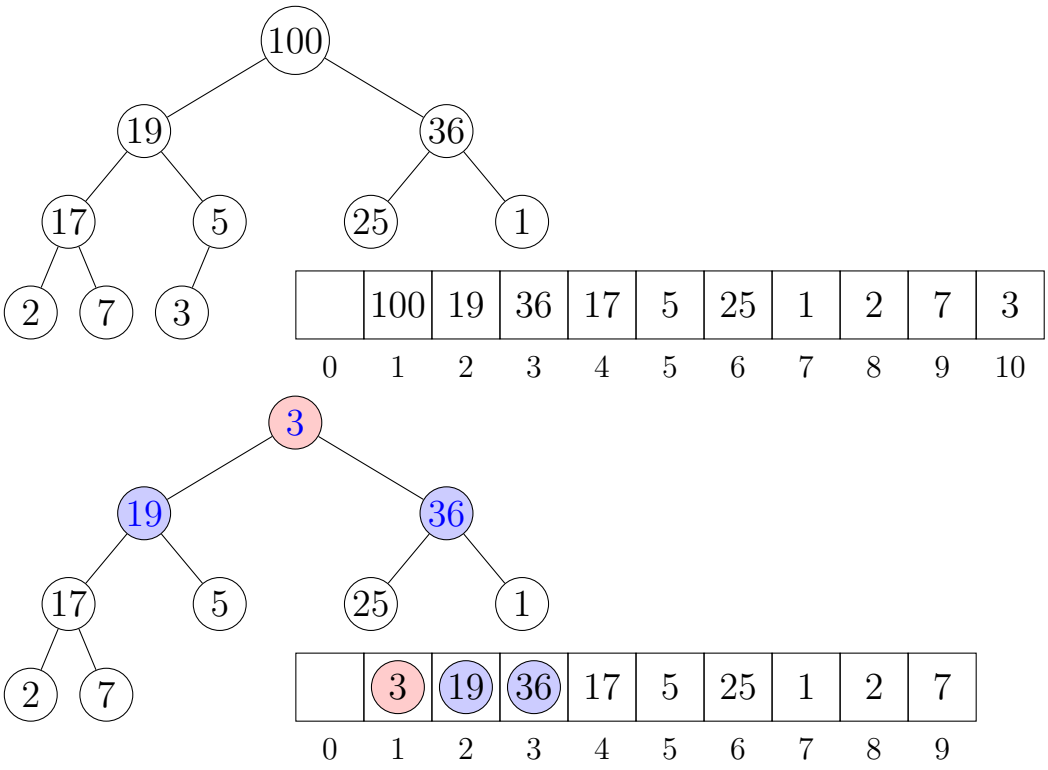


Figure 3.3: [Top] Heap before extracting the maximum. [Bottom] Heap after removing 100, and moving 3 the top. The heap invariant is violated; `sift_down` needs to swap 3 with $\max\{19, 36\}$.

3.2.4 The `extract` function

Since the minimum element (in case of min-heap, or maximum element in case of max-heap) is already at the top of the tree we remove it from the heap and return it. However, doing so will leave a cap at the top of the heap, so we need to fill the gap. The conventional way to fill the gap is to simply replace it with the right-most leaf. However, doing so likely violates the invariant of the heap, the new top of the heap is no longer the minimum element. This can be fixed by calling the `sift_down` function.

As an illustration, suppose we begin with the heap shown in Figure 3.3 [Top], and invoke `extract`. The node at the top, 100, is removed (and eventually returned from the function), but first we must repair the heap because of the gap. We delete 3, the right-most element, and fill the hole at the top of the heap with 3, as shown in Figure 3.3 [Bottom]. Now the heap invariant is violated for two reasons: $3 \geq 19$ is false and

also $3 \geq 36$ is false. A call to `sift_down` will fix this. See Section 3.2.7.

3.2.5 The `heapify` function

A naive (inefficient) implementation of this function would simply create an empty heap by calling `create_heap`, and then call `insert` for each of the elements in the given list. However, this can be done more efficiently as follows.

The given list might accidentally (extremely unlikely but possible) already be in the correct order. So the idea, is simply *fix* the elements which are wrong. This turns out to be faster than rebuilding the heap from scratch.

What we must do is ensure the heap invariant on every element of the heap. For this explanation, assure we are using a max-heap. The heap invariant is that every parent must be greater than all children, and there are zero, one, or two children. This means the heap invariant is already satisfied on all nodes with no children; *i.e.* the invariant is satisfied on all leaves. Thus we only need to enforce the variant on internal nodes (*i.e.* half the tree).

The first node to visit, is the first parent which as a child. If there are n-many nodes, then there are parent of the right-most child is the right-most parent. So we simply compute its index as $\lfloor \frac{i}{2} \rfloor$ (in the case of a one-based implementation) or $\lfloor \frac{i-1}{2} \rfloor$ (in the case of a zero-based implementation). We must examine the nodes starting at the right-most parent in the heap, and traverse the heap in reverse order back to the beginning (index 0 or 1). At each step, we may need to swap the parent and child to enforce the invariant. If we perform a swap, then the subtree might now be invalid, so we need to walk down that tree. The `sift_down` function performs this downward walk. Then we simply call `sift_down` on all parents with children, but traverse in reverse order; *i.e.* traverse from i down to 0.

```
# assuming we are using a 1-based encoding
for parent in range(len(data) // 2, 0, -1):  # skipping index 0
  sift_down(h, parent)
```

Since each call to `sift_down` does a partial walk of a single branch of the tree, the `heapify` operation does $\frac{n}{2} \log n$ or fewer swaps.

3.2.6 The `sift_up` function

After a new element has been added to the heap by simply adding it to the end (at the right-most position) as shown in Figure 3.4 [Top], or during a call to the `heapify` function, the heap variant might be violated. In this case we observe that $5 \not\geq 42$. Attempting to fix this violation, we swap 5 and 42 as shown in Figure 3.4 [Middle]. How we find the invariant is again violated because $19 \geq 42$ is false, so we continue sifting up as shown in Figure 3.4 [Bottom]. Since the invariant $100 \geq 42$ is satisfied, we can terminate the walk. The heap variant has been restored.

Since `sift_up` only walks up one branch of the tree, and possibly short-circuits part of the way up, the complexity of this operation is never more than $\log n$ where n is the element count in the heap.

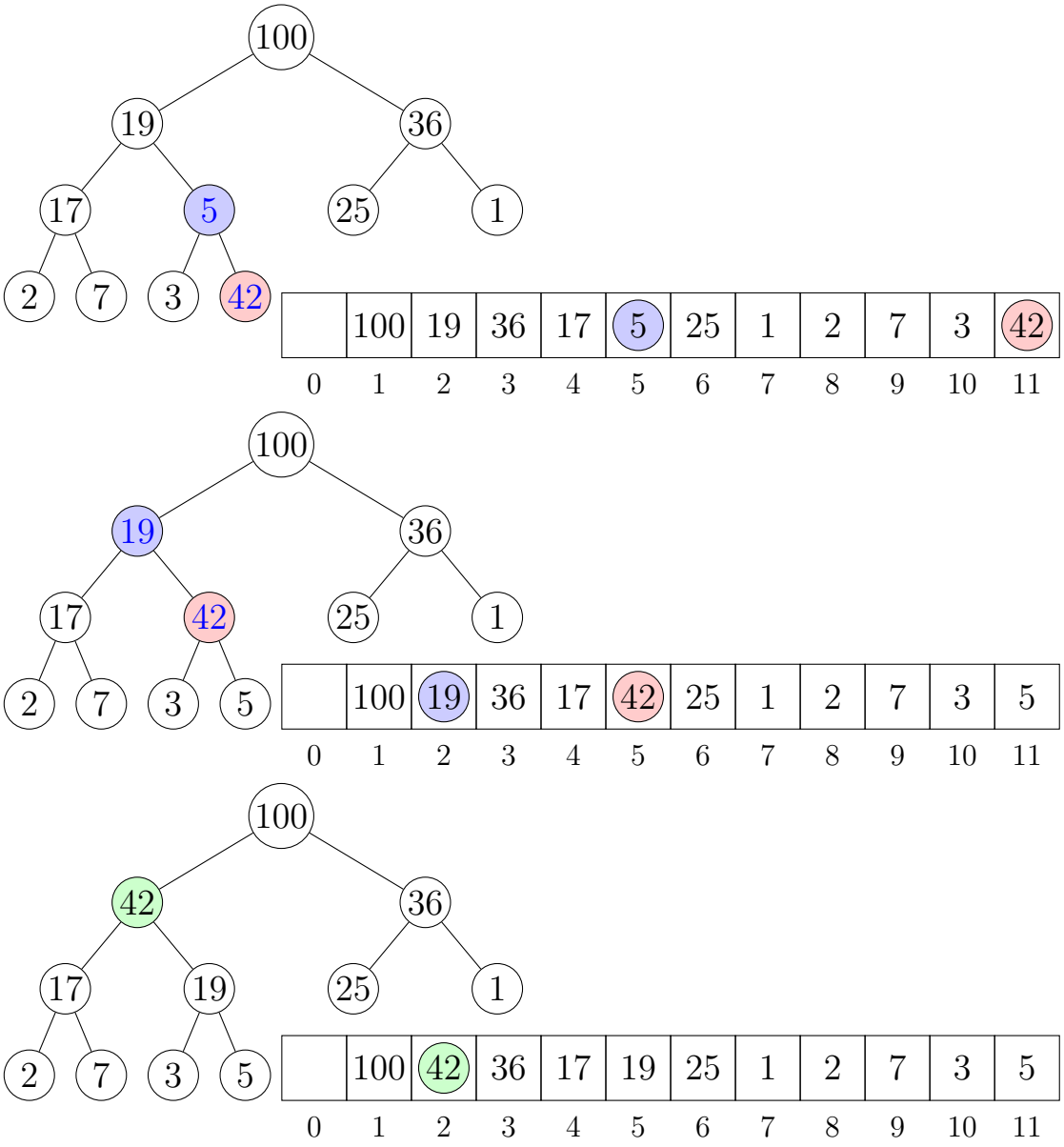


Figure 3.4: [Top] Inserting 42 in right-most slot of array. [Middle] Sifting up, swapping 5 and 42. [Bottom] Sifting up, swapping 42 and 19.

3.2.7 The `sift_down` function

After an element has been moved from the right-most position in the heap to the top of the heap as shown in Figure 3.5 [Top], the heap variant might be violated. In this case we observe two violations $3 \not\geq 19$ and $3 \not\geq 36$. Attempting to fix this violation, we examine the two children of 3 and select the larger of the two, *i.e.*, 36. We swap 36 and 3 and find yet another violation as shown in Figure 3.5 [Middle].

In particular $3 \not\geq 25$. So we walk further down the tree swapping 3 and 25, resulting in Figure 3.5 [Bottom], and find that the heap invariant is now restored: every parent node in the heap is greater than (or equal to) all its children.

When sifting down, there are several cases to consider.

1. The parent has no children, then stop the walk.
2. The parent has one child, necessarily a left-child. Because of how the heap is constructed, it is impossible to have a right-child without a left-child. If the parent is greater than the left-child, then stop. Otherwise, swap and continue the walk in the left-child.
3. The parent has two children. If the parent is greater than the maximum child, then stop the walk, otherwise swap parent with maximum child and continue the walk into that child.

Since `sift_down` only walks down one branch of the tree, and possibly short-circuits part of the way down, the complexity of this operation is never more than $\log n$ where n is the element count in the heap.

3.2.8 The `heapq` library

The Python standard library contains an implementation heap implementation called `heapq` which you should use in the Huffman Tree exercises in Section 3.3. Three important functions in that library are `heqpq.heappush`, `heapq.heappop`, and `heapq.heapify`.

```
1  from heapq import heappush, heappop, heapify
```

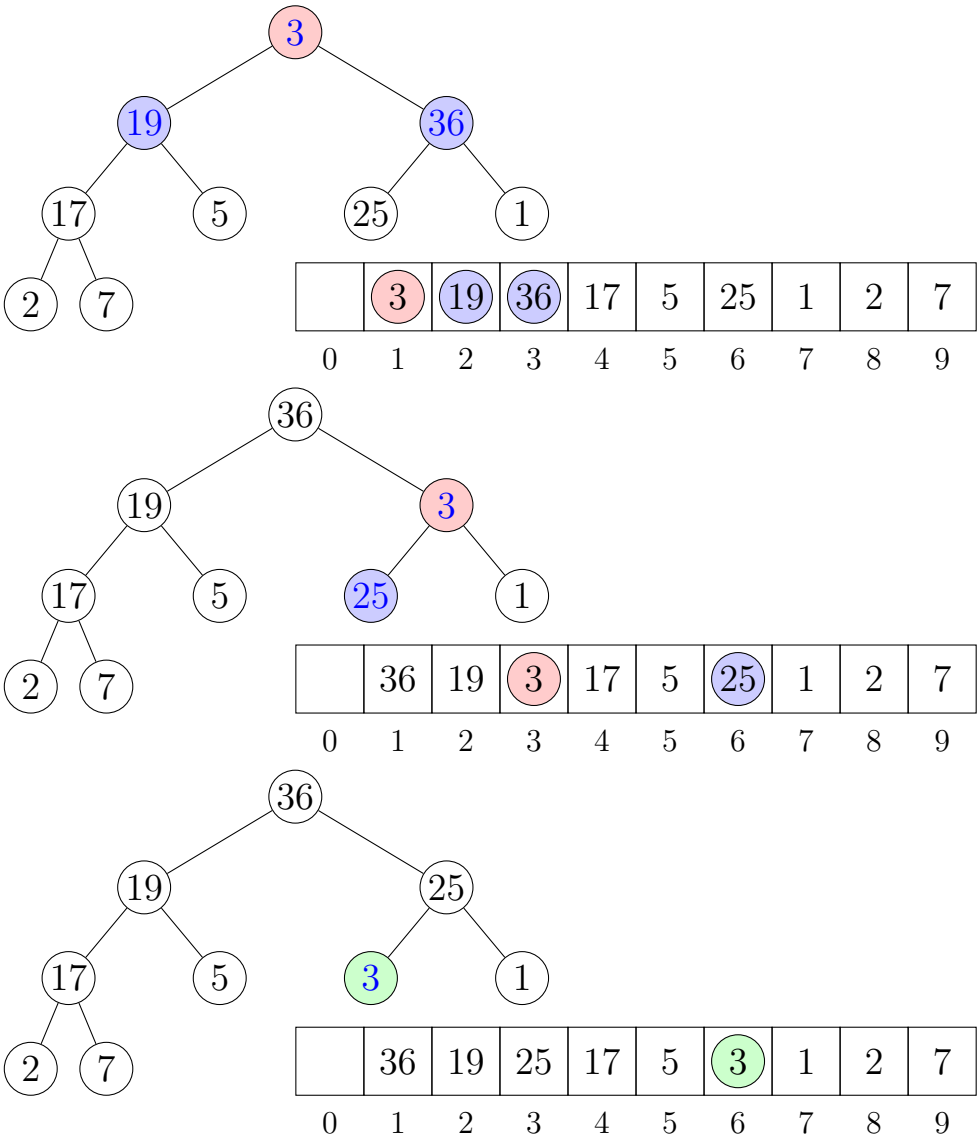


Figure 3.5: [Top] Maximum removed and replaced with right-most element, 3. [Middle] Sifting down, swapping 3 and 36. [Bottom] Sifting down, swapping 3 and 25.

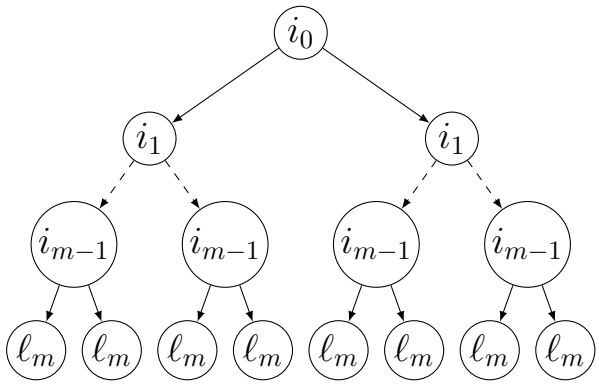


Figure 3.6: Full binary tree with n nodes and $m = 3$ levels, where $m = (\log_2 n) - 1$.

3.2.9 Complexity of Heaps

What is the worst case complexity of the `heapify` function? The worst case arises when `sift_down` is called on all the internal nodes and if in each case `sift_down` must perform the maximum numbers of swaps. Consider the full binary shown in Figure 3.6.

If there are n nodes in the binary tree then there are approximately $\frac{n}{2} = 2^m$ leaf nodes and $\frac{n}{2}$ internal nodes. There is actually one more leaf node than internal node, but for large values of n we ignore this discrepancy. In Figure 3.6 we have $15 \approx 16 = 2^4$ total nodes, with $8 = 2^3$ leaf nodes.

The `heapify` function will call `sift_down` on all the nodes in level $m-1$, and then on all the nodes in level $m-2$, and so on up to level 0. There are $\frac{n}{4}$ nodes in level $m-2$, $\frac{n}{8}$ in row $m-3$, up to 1 node in level 0. When `sift_down` is called on each element of level $m-1$, the content of the node is swapped with at most one level below it; *i.e.* $\frac{n}{4}$ swaps. When `sift_down` is called on each element of level $m-2$, the content of the node is swapped with at most two levels below it, but there are half as many nodes at level $m-2$ as compare to level $m-1$; this means there are at most $2 \times \frac{n}{8} = \frac{n}{4}$ swaps.

For level $m-3$ there are $\frac{n}{8}$ nodes, and each call to `sift_down` must swap at most three times; *i.e.*, $3 \times \frac{n}{8}$. Figure 3.7 contains a summary of this process.

Thus the total number of swaps is the sum of the right-most column of

level	num nodes	max swaps per sift_down	max swaps per level
$m-1$	$\frac{n}{2}$	1	$1 \times \frac{n}{2}$
$m-2$	$\frac{n}{4}$	2	$2 \times \frac{n}{4}$
$m-3$	$\frac{n}{8}$	3	$3 \times \frac{n}{8}$
$\vdots$			$\vdots$
$m-k$	$\frac{n}{2^k}$	k	$k \times \frac{n}{2^k}$
$\vdots$			$\vdots$
0	1	m	m

Figure 3.7: Worst-case swaps per level per call to function `heapify`

the table in Figure 3.7. $\sum_{k=0}^{m-1} k\frac{n}{2^k}$ where $m = (\log_2 n) - 1$. In Claim 3.2.3, we compute an upper bound of the summation: $\sum_{k=0}^{\log_2 n} k\frac{n}{2^k}$. To compute this upper bound, we will make use of two intermediate results: Claim 3.2.1 and Claim 3.2.2.

Claim 3.2.1

If $|x| < 1$ then $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$.

Proof. Let $S_n(x) = \sum_{k=0}^{n} x^k$. First note that $S_n(x) - 1 = \sum_{k=1}^{n} x^k$ simply by omitting the $k = 0$ term of the sum.

$$\begin{aligned} xS_n(x) = x\sum_{k=0}^{n} x^k &= \sum_{k=0}^{n}(x)(x^k) \\ &= \sum_{k=0}^{n} x^{k+1} = \sum_{k=1}^{n+1} x^k \end{aligned}$$

Now

$$\begin{aligned}
S_n(x) - xS_n(x) &= \sum_{k=0}^{n} x^k - \sum_{k=1}^{n+1} x^k \\
&= \underbrace{x^0}_{k=0} + \sum_{k=1}^{n} x^k - \left(\sum_{k=1}^{n} x^k + \underbrace{x^{n+1}}_{k=n+1} \right) \\
&= \underbrace{\sum_{k=1}^{n} x^k - \sum_{k=1}^{n} x^k}_{=0} + x^0 - x^{n+1} \\
&= 1 - x^{n+1} \\
S_n(x)(1-x) &= 1 - x^{n+1} \\
S_n(x) &= \frac{1 - x^{n+1}}{1-x}
\end{aligned}$$

So now we have the finite sum, but we need the infinite sum.

$$\begin{aligned}
S(x) = \lim_{n\to\infty} S_n(x) &= \sum_{k=0}^{\infty} x^k \\
&= \lim_{n\to\infty} \frac{1 - x^{n+1}}{1-x} \\
&= \lim_{n\to\infty} \frac{1}{1-x} - \lim_{n\to\infty} \frac{x^{n+1}}{1-x} \\
&= \frac{1}{1-x} - \frac{1}{1-x} \underbrace{\lim_{n\to\infty} x^{n+1}}_{\to 0, \text{ since } |x|<1} \\
&= \frac{1}{1-x} - \frac{1}{1-x}(0) \\
&= \frac{1}{1-x}
\end{aligned}$$

We use Claim 3.2.1 to prove Claim 3.2.2.

Claim 3.2.2

If $|x| < 1$ then $\sum_{k=0}^{\infty} kx^k = \frac{x}{1-x}$.

Proof. As in Claim 3.2.1, let $S(x) = \sum_{k=0}^{\infty} x^k$.

$$S(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{by Claim 3.2.1.}$$

$$\begin{aligned}
\frac{d}{dx}S(x) &= \frac{d}{dx}\sum_{k=0}^{\infty} x^k = \frac{d}{dx}\frac{1}{1-x} \\
&= \sum_{k=0}^{\infty} \frac{d}{dx}x^k = \frac{d}{dx}(1-x)^{-1} \\
&= \sum_{k=0}^{\infty} kx^{k-1} = (-1)(1-x)^{-1}(-2) = \frac{1}{(1-x)^2} \\
x\frac{d}{dx}S(x) &= \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}
\end{aligned}$$

Now we tackle the upper bound of the sum, $\sum_{k=0}^{m-1} k\frac{n}{2^k}$; *i.e.* the rightmost column in Figure 3.7.

Claim 3.2.3

$$\sum_{k=0}^{m-1} k\frac{n}{2^k} < 2n$$

Proof.

$$\begin{aligned}
\sum_{k=0}^{m-1} k\frac{n}{2^k} &= n \sum_{k=0}^{(\log_2 n)-2} \frac{k}{2^k} \\
&< n \sum_{k=0}^{\infty} \frac{k}{2^k} \\
&= n \sum_{k=0}^{\infty} k \left(\frac{1}{2}\right)^k \\
&= n \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} && \text{letting } x = \frac{1}{2} \text{ in Claim 3.2.2} \\
&= n \frac{1/2}{1/4} = 2n
\end{aligned}$$

Therefore, we see that the worst case number of swaps performed by a call to `heapify` on an array of size n fewer than $2n$. We denote this with so-called big-O notation as $O(n)$.

3.2.10 Homework

As a homework assignment you must implement the functions described in Section 3.2.1.

There is a homework assignment `homework/heap.py`; the test case is found in `tests/heap_test.py`. The student should implement the functions which are incomplete in this file.

3.3 Huffman Trees

Huffman encoding is a type of lossless data compression.

A Huffman code is a prefix code. A *prefix code* is a mapping from letter to binary code such that no code is the prefix of another code. The following code is *not* a valid prefix code, because, for example the code for `a=11` is a prefix of the code `c=110`.

a $\leftrightarrow$ 11
b $\leftrightarrow$ 10
c $\leftrightarrow$ 110
d $\leftrightarrow$ 0

As a consequence, using this invalid code, we would be able to decode the message `10100` either as `10 11 0 = b a d` or as `10 110 = b c`.

In order to understand the motivation of building a Huffman code, we need to first understand how decoding works. We will discuss both the decoding phase (Section 3.3.1) and the encoding phase (Section 3.3.4).

The ultimate goal of selecting a Huffman code is to minimize the size of the encoded message. But there's a small complication. In order to decode an encoded message the recipient must have access to the code itself, *i.e.*, the correspondence of letter (`a`, `b`, `c` to binary sequence `0`, `100`, `101`). Thus to send an encoded message, the message just also contain enough information to perform the decoding. One such encoding of the code itself is discussed in Section 3.3.6.

Since the serialized code must be included in the message, it is possible in the worst case, that the encoded message is actually longer than the original message.

Since the encoded file will contain the table portiona and also the encoded message portion, there must be some way for the decoder to determine where the seperator is. So there is yet another piece of information necessary, the delimeter or at least a convention for detecting where the table stops and the message begins.

We can compute the size of the encoded message given the letter frequency and the code. Given the code in Figure 3.8, Suppose we have

a 150 character message which contains the letters `a` through `f` with the frequencies: `a` 75 times, `b` 36 times, `c` 20 times, `d` 10 times, `e` 5 times, `f` 4 times. We can compute the number of bits necessary for the encoded message portion (not including the table portion).

$$\underbrace{(75)(1)}_{a} + \underbrace{(36)(3)}_{b} + \underbrace{(20)(3)}_{c} + \underbrace{(10)(3)}_{d} + \underbrace{(5)(4)}_{e} + \underbrace{(4)(4)}_{f} = 309 \text{ bits}$$

We need 309 bits, but since memory comes in blocks of 1 byte = 8 bits, then we need $\lceil \frac{309}{8} \rceil = 39$ bytes, to contain the encoded message.

It should be clear from this simple example choose a good prefix code, we need to assign symbols of high frequency to shorter codes, and lower frequency symbol to longer codes. But there are many ways of doing this. The question is what is the best assignment of symbols to lengths to minimize the size of the encoded message.

Symbol	Code word	Code length
a	0	1
b	100	3
c	101	3
d	110	3
e	1110	4
f	1111	4

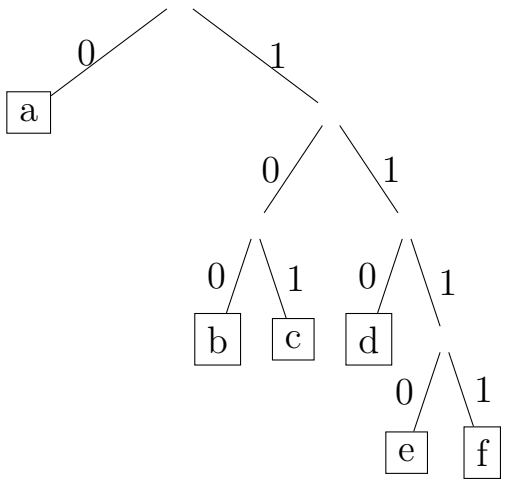


Figure 3.8: Left: Huffman Codewords. Right: Huffman Tree.

3.3.1 Prefix Code Decoding

Decoding a message which as been encoded using the Huffman encoding (or any other prefix code), can be implemented as a binary tree traversal. The first step is to build the binary tree given the encoding. Figure 3.8 shows an encoding and the corresponding tree. Each code, `0`, `100`, ..., `1111` corresponds to a different complete branch from the root of the tree to a leaf.

Once the tree has been build, message decoding corresponds to repeated tree traversal. For example to decode the following message `101010011111110110`. The first question one asks is: How do we know when the code for one letter ends and another begins. The answer is because this is a prefix code, there is only one correct parsing; *i.e.*, the binary sequence can only be interpreted one way.

Start at the top of the tree, and simply follow the branches left or right depending on whether a 0 or 1 is read.

1. Follow the branches 1, 0, 1 to find the leaf `c`. So we output the character `c`.
2. Start over at the root to decode 0, finding a leaf `a`, so output character `a`.
3. Start over at the root and decode 1, 0, 0, finding leaf `b`, so output character `b`
4. Start over at the root and decode 1, 1, 1, 1, finding leaf `f`, so output character `f`.
5. Start over at the root and decode 1, 1, 1, 0, finding leaf `e`,
6. Start over at the root and decode 1, 1, 0, finding leaf `d`, so output character `d`.

The final decoded message is `cabfed`.

$$\underbrace{1\,0\,1}_{c}\ \underbrace{0}_{a}\ \underbrace{1\,0\,0}_{b}\ \underbrace{1\,1\,1\,1}_{f}\ \underbrace{1\,1\,1\,0}_{e}\ \underbrace{1\,1\,0}_{d}$$

3.3.2 Prefix Code Decoding Example

Given the prefix code in Figure 3.9 let's encode the message:
[`she sells sea shells`]. The message has the frequencies shown in Figure 3.9.

Looking at the frequencies, we see that we should associate the smallest code with `s` and the longest code with `a`. Furthermore whenever two symbols, (s_1 and s_2) have different frequencies: say $f_1 \leq f_2$ then the

length of the codes must obey the opposite order $|s_1| \geq |s_2|$. There are many ways of satisfying these constraints. Figures 3.9 and 3.10 illustrate two alternatives.

Symbol	fequency	Code
s	6	00
e	4	01
l	4	100
space	3	101
h	2	110
a	1	111

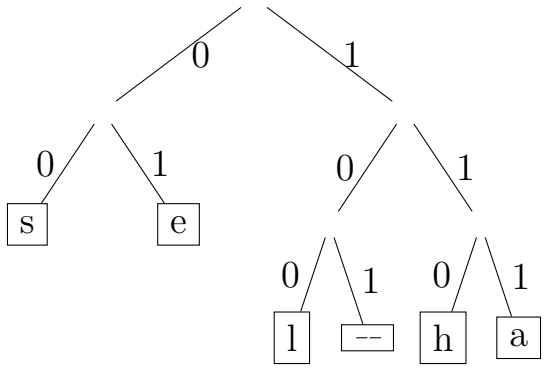


Figure 3.9: Left: Prefix Code Code Words. Right: Prefix Code Tree.

With the first encoding (Figure 3.9), the message, `she sells sea shells`, is encoded into 50 bits as follows.

```
she_    00 100 01 101
sells_  00 01 100 100 00 101
sea_    00 01 111 101
shells  00 110 01 100 100 00
```

However, this would be regrouped into 7 8-bit bytes as follows.

```
00100011
01000110
01000010
10001111
10100110
01100100
00
```

With the second encoding (Figure 3.10), the same message, `she sells sea shells`, is encoded into 49 bits as follows.

```
she_     0 1110 100 110
sells_   0 100 101 101 0 110
sea_     0 100 1111 110
shells_  0 1110 1110 100 0
```

Again, this would be regrouped into 6 8-bit bytes as follows.

Symbol	Code
s	0
e	100
l	101
space	110
h	1110
a	1111

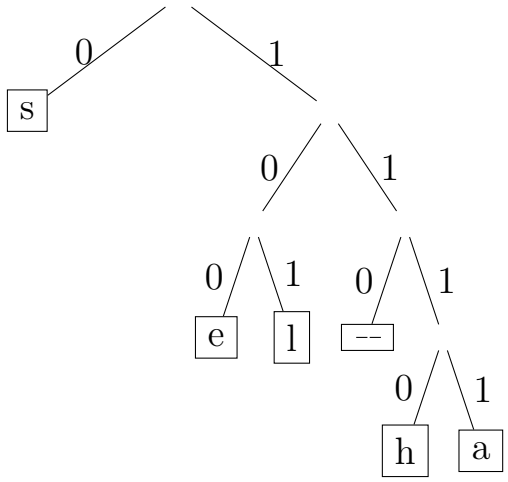


Figure 3.10: Left: Prefix Code Codewords. Right: Prefix Code Tree.

```
01110100
11001001
01101011
00100111
11100111
01110100
0
```

We see that the second encoded results in one fewer bits of encoded message, but the same number of bytes. The longer the message, in general, the more difference we would see in the two results.

What is the commonality connecting the binary trees in Figures 3.9 and 3.10? It is the fact that they have 6 leaves. Any binary tree with n leaves induces a prefix code for n symbols. We simply distribute the symbols amongst the leaves so that the depth leaf corresponds inversely to the frequency of the symbol. *I.e.*, the most frequent symbol goes to the deepest leaf, and the least frequent symbol goes at the leaf closest to the root.

3.3.3 Counting Prefix Codes

Given that every binary tree with n leaves corresponds to a prefix code for n symbols, we'd like to know how many prefix codes exist which respect the proposition $f_i \leq f_j \iff |s_i| \geq |s_j|$, where f_i is the frequency of

symbol s_i and $|s_i|$ is the length of the code corresponding to symbol s_i.

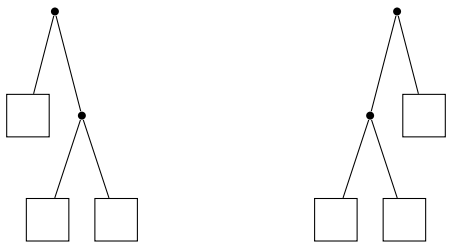
For the purpose of counting, we say that a binary tree with every internal node having exactly two children. How many binary trees exist with n leaves? The *Catalan numbers* count the number of binary trees. The nth Catalan number C_n counts how many (unlabeled) binary trees have n internal nodes—equivalently, $n + 1$ leaves.

$$C_n = \frac{1}{n+1}\binom{2n}{n} \tag{3.1}$$

The several Catalan numbers are:

n+1 leaf nodes	**n internal nodes**	C_n
1	0	1
2	1	1
3	2	2
4	3	5
5	4	14
6	5	42
7	6	132
8	7	429
9	8	1430
10	9	4862
15	14	2674440
20	19	1767263190

The formula in Equation (3.1) considers two isomorphic binary trees as being different trees. Two trees are considered isomorphic if one can be transformed to the other simply by swapping right and left children.



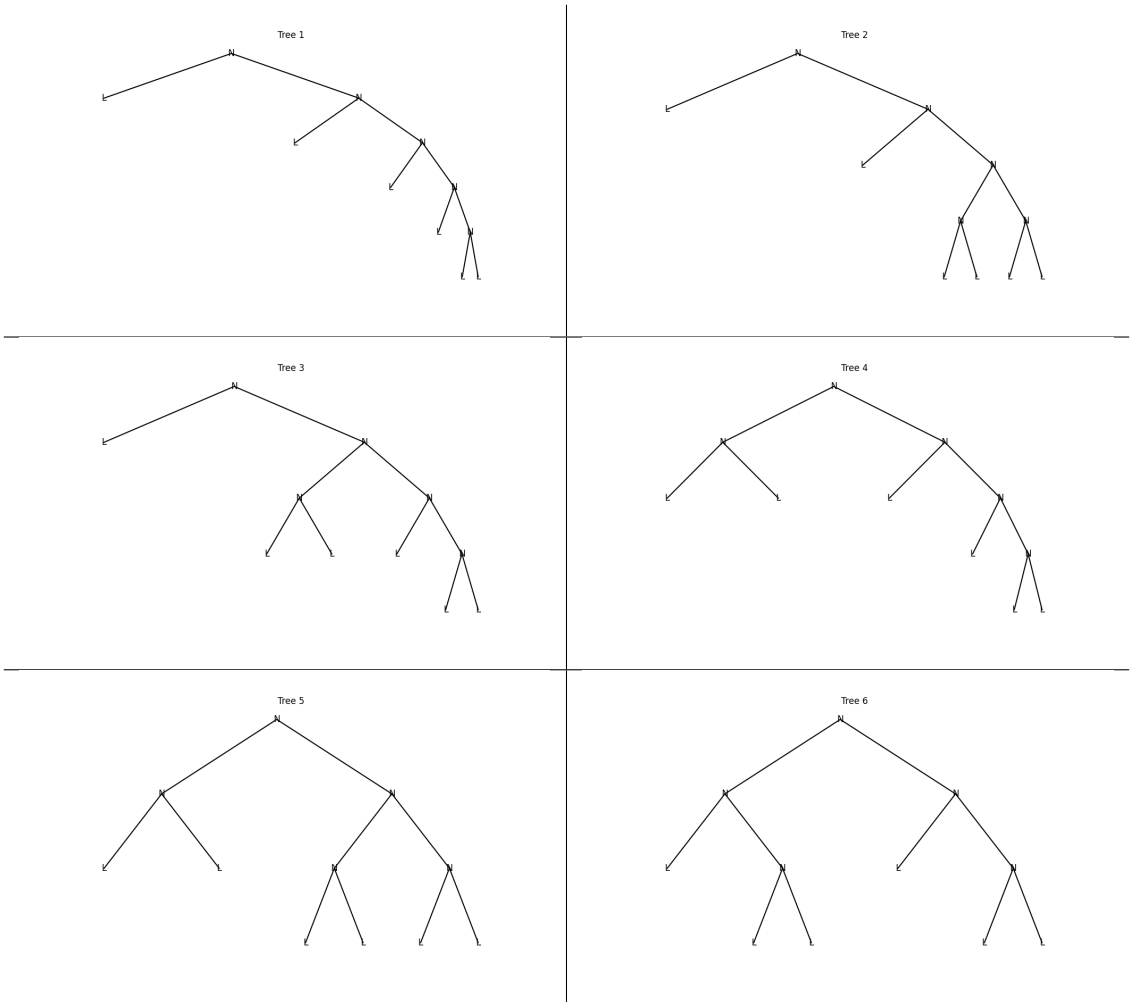


Figure 3.11: 6 Non-isomorphic 6-leaf binary trees

Even though there are 42 unlabeled full binary trees with 6 leaves, there are only 6 non-isomorphic such trees. These trees can be seen in Figure 3.11.

The number of such trees can be counted using the formula:

$$a_n = \sum_{k=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \begin{cases} \frac{1}{2} a_k \left(a_k + 1 \right) & \text{if } k = n - 1 - k \\ a_k a_{n-1-k} & \text{otherwise} \end{cases}$$

Even though there are 6 non-isomorphic full binary trees as seen in Figure 3.11, not all of them result in different prefix codes. For example, in the figure, the two in the bottom row both induce a prefix code having 2 codewords of length 2, and 4 codewords of length 3.

3.3.4 Huffman Encoding

We see in Equation (3.1) that there are many valid prefix codes for a given message. However, which code gives the best (shortest) encoding? For a given message, with a fixed frequency of characters, the Huffman code is a prefix code which encodes the message to the minimum length.

We can generate the Huffman code by generating the Huffman tree.

We would like to generate the encoding (and tree) shown in Figure 3.8. We can generate the encoding given the frequency of letters, either as raw occurrence counts or as percentages.

To build the Huffman tree, we start with a collection of singleton trees (each tree with a single leaf node and no internal nodes), each corresponding to a symbol and the corresponding count (or percentage).

The algorithm is:

1. Take the two trees, t_1 and t_2 with the smallest counts.
2. Remove the two trees from the collection.
3. Create a new tree by creating a new root node, having the t_1 and t_2 as left and right child nodes.
4. Associate a count with this new node which is equal to the sum of the counts of the two child nodes.
5. Add this new tree to the collection.
6. Repeat from step 1 until only one tree remains.

The tree that remains when only one tree is remaining in the heap is the Huffman tree.

It is an interesting implementation detail that inn each iteration, if at least two trees are remaining, we select the two with the lowest count. This means we can use the min-heap data structure discussed in Section 3.2, where the priority used is simply the count. Taking the minimum element correspond to calling the `extract` function on the heap. Adding the newly formed element to the heap correspond to calling the function `insert`.

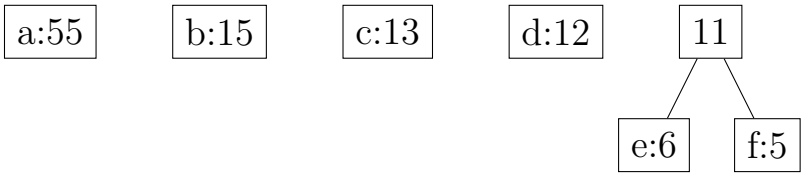
Symbol	Occurrence Count	Percentage
a	55	51.9
b	15	14.2
c	13	12.3
d	12	11.3
e	6	5.7
f	5	4.7
total	106	100%

Figure 3.12: Symbols and Occurrences as input to Huffman Encoding Algorithm

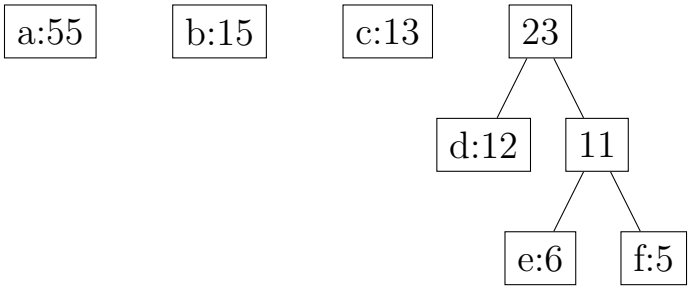
3.3.5 Example of Huffman Algorithm

Let's iterate through the symbols and counts from the table in Figure 3.12.

We find the two lowest counts. *I.e.*, this corresponds to two calls to `extract` from a min-heap. We get $f : 5$ and $e : 6$, we form a new tree with priority $5 + 6 = 11$ with $f : 5$ and $e : 6$ as child nodes.

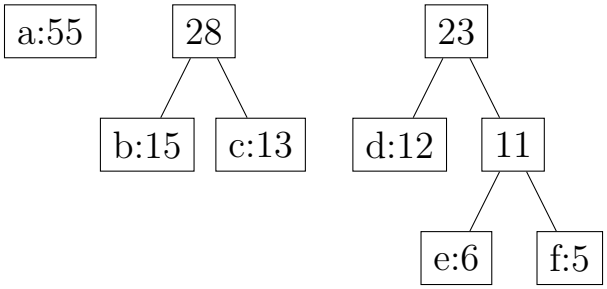


We find the two lowest counts (call `extract` twice), giving us 11, and 12. We create a new root with the two trees and left and right child. The new root has priority $11 + 12 = 23$.

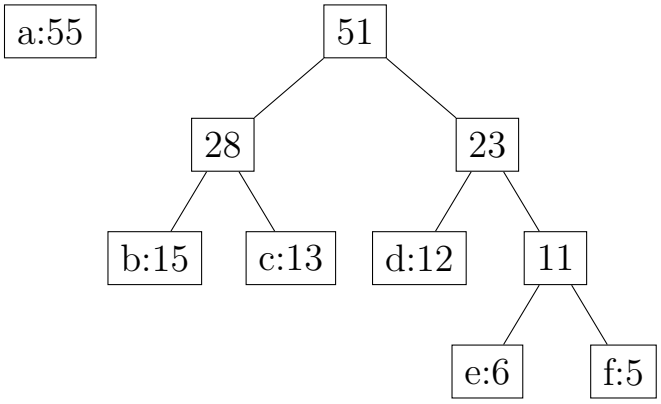


Again find the two lowest counts: $c : 13$ and $b : 15$. Remove them,

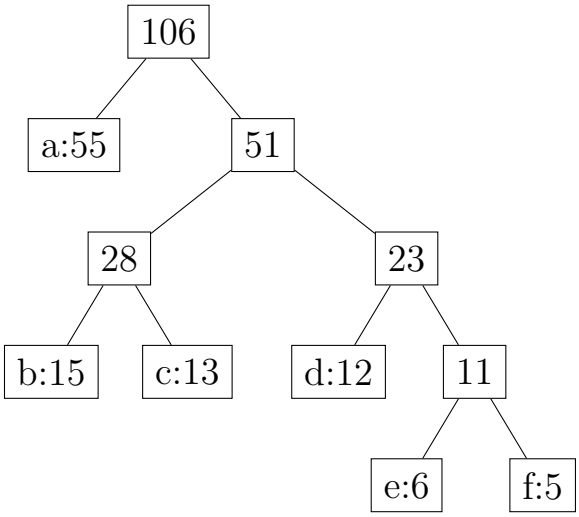
and add a new root with $c : 13$ and $b : 15$ as child nodes. The priority of the new tree is $13 + 15 = 28$.



Again find the two lowest counts: 23 and 28. Remove them, and add a new root with 28 and 23 as child nodes. (It does not matter which is left and which is right). The priority of the new tree is $23 + 28 = 51$.



Finally, the two lowest counts, are actually the final two trees: 51 and $a : 55$. Remove them and build a new root with priority $51 + 55 = 106$ having the two trees as left and right children.



We can now label the edges 0 and 1. It does not matter which, but to achieve the encoding from Figure 3.8, we label 0 each branch to a left child, and label 1 each branch to a right child.

3.3.6 Serializing the Huffman Tree

To successfully compress a file using Huffman encoding, we must create a new file with the concatenated sequence of codes for each symbol in the original file. But we must also encode the Huffman tree. Otherwise, it will be impossible to decode the file later after the encoding (and Huffman tree) have been forgotten.

We would like the encoding of the tree itself to be as small as possible, otherwise the resulting file could end up being larger than the original file. To encode the tree, we first express it as a hierarchical list where leaf nodes are wrapped in square brackets `[a]`, `[b]`, etc, and internal nodes are expressed as pairs (as every internal node has exactly two child nodes).

Thus we can express the Huffman tree shown above (which is a duplicate of the three in Figure 3.8), as the following hierarchical object.

```
([a], (([b], [c]),
       ([d], ([e],
              [f]))))
```

Next, we replace each character with its 8-bit ASCII code. `a-f` correspond to ASCII codes 141 through 146. The corresponding 8-bit binary codes are 01100010, 01100011, ..., 01100110. Of course if all the symbols in the file are between 0 and 127, we could just as well use the 7-bit ASCII codes. The tree encoding becomes.

```
([0110 0010], (([0110 0011], [0110 0100]),
               ([0110 0101], ([0110 0110],
                              [0110 0111]))))
```

Next, since we know pairs have two elements, and leaves have single elements, we can omit the commas, closing parentheses and closing brackets.

```
([0110 0010 (([0110 0011 [0110 0100
              ([0110 0101 ([0110 0110
                            [0110 0111
```

Finally, we convert open parenthesis to 0, and open square bracket to 1.

```
01 0110 0010 001 0110 0011 1 0110 0100
             01 0110 0101 01 0110 0110
                           1 0110 0111
```

Now, the spaces and new-line characters are irrelevant. The tree can be encoded as the following sequence of $7 \times 8 + 3 = 59$ bits.

```
01011000
10001011
00011101
10010001
01100101
01011001
10101100
111
```

To rebuild the tree from this sequence of 59 bits, just perform the operations in reverse order.

3.3.7 Homework

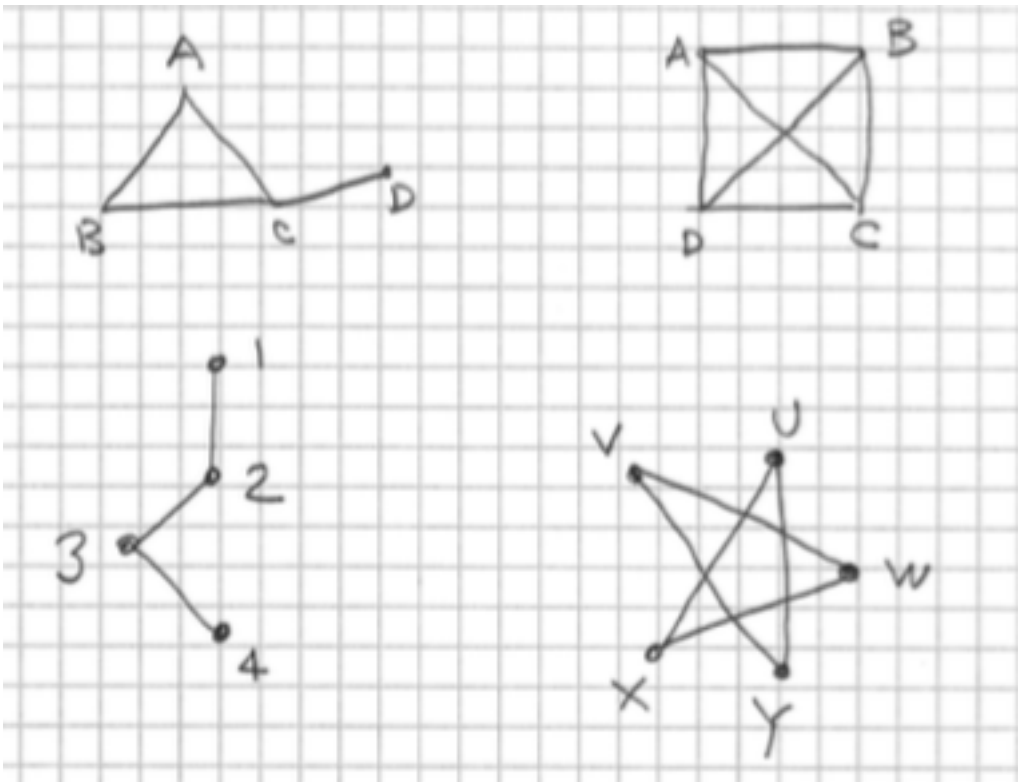
There is a homework assignment `homework/huffman.py`; the test case is found in `tests/huffman_test.py`. The student should implement the functions which are incomplete in this file.

Chapter 4

Graphs

Chapter Objectives

- Define a simple graph (V, E): vertex set, edge set, adjacency, incidence, degree; distinguish undirected from directed and DAG from general directed graph.
- Represent a graph as an adjacency list in $\Theta(|V|+|E|)$ space and state when this is preferable to an adjacency matrix.
- Trace DFS on an arbitrary graph, explaining why a visited set prevents infinite loops when the graph contains cycles.
- Apply Kahn's algorithm (1962): maintain a set S of in-degree-zero nodes, detect a cycle when S empties with edges remaining, and apply a tie-breaker for deterministic output.
- Apply Tarjan's DFS-based topological sort and explain why a topological numbering produces a lower-triangular adjacency matrix.
- Schedule dependent jobs sequentially or on two threads using a topological ordering with a slowest-first tie-breaking heuristic.
- Count k-step walks from v_1 to v_2 as the (v_1, v_2) entry of A^k, computed via repeated matrix multiplication.



Intuitively, graphs are collections of vertices and connections between them.

Note that crossing lines are *not* connections.

4.1 Kinds of Graphs

Definition 4.1.1 (*k-subsets*)

If V is a set of n elements, ($|V| = n$), then the *k-subsets* are the subsets of V having exactly k elements. We denote the set of k-subsets as $\binom{V}{k}$.

$$\binom{V}{k} = \{u \subset V \mid |u| = k\} = \{u \in 2^V \mid |u| = k\}$$

Note that the size of $\binom{V}{k}$ is $\left|\binom{V}{k}\right| = \binom{|V|}{k} = \binom{n}{k} = \frac{n!}{k!(n-k)!}$.

We are particularly interested in $k = 2$, i.e., the $2 - subsets$, $\binom{V}{2}$.

$$\left|\binom{V}{2}\right| = \frac{|V|!}{2!(|V|-2)!} = \frac{n \times (n-1)}{2}$$

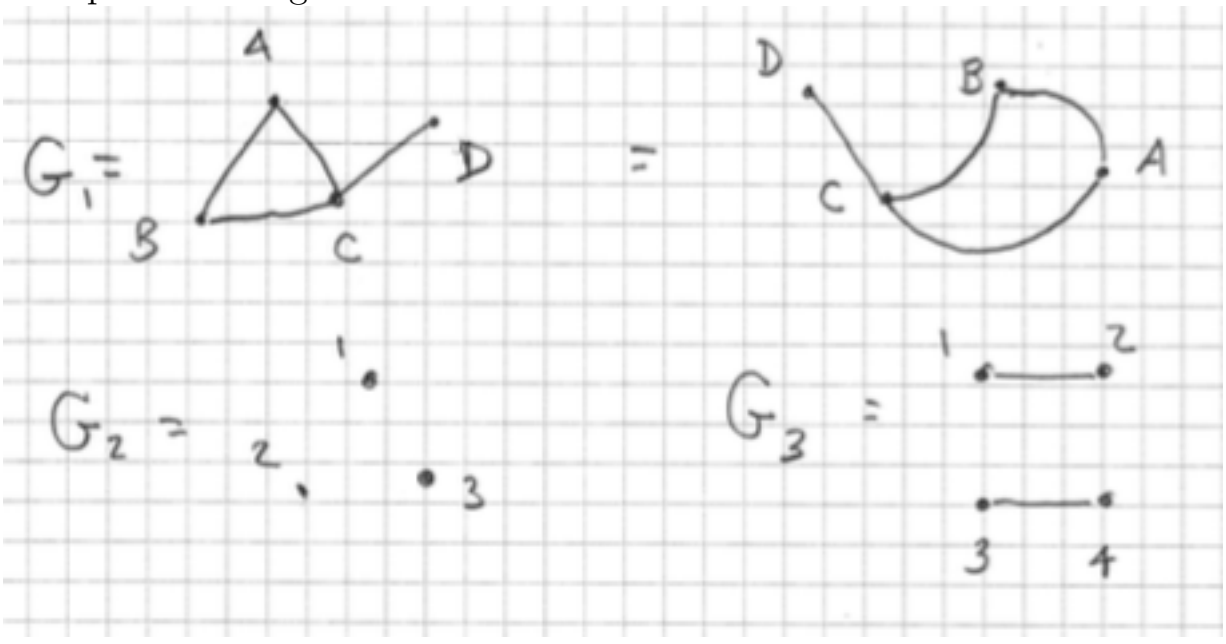
Definition 4.1.2 (*simple graph*)

A *simple graph*, (V, E), is an object consisting of two finite sets V and E, such that $V \neq \emptyset$, and $E \subset \binom{V}{2}$. V is called the *vertex set*, and each element of V is called a *vertex* or a *node*. E is called the *edge set*, and every element of E is called an *edge* or an *arc*.

Examples:

$$G_1 = \big(\{A, B, C, D\}, \{\{A, B\}, \{B, C\}, \{A, C\}, \{C, D\}\}\big)$$
$$G_2 = \big(\{1, 2, 3\}, \emptyset\big)$$

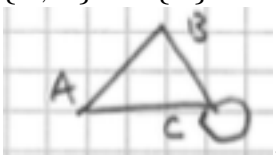
We can naturally represent graphs by drawing them. But a graph does not depend on the arbitrarily chosen "position" of its drawn vertices nor "shape" of its edges.



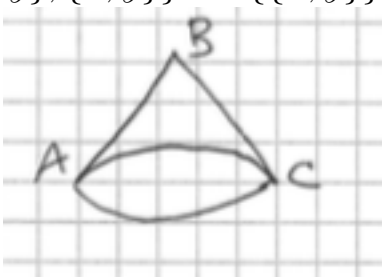
Some consequences of the definition of simple graph:

- A graph is not necessarily connected. E.g., G_3.

- E may be empty.
- V is never empty.
- Edges have no "direction", because $\{x, y\} = \{y, x\}$. We may also talk about *directed graphs*, but these are not simple graphs.
- There are no looping edges because $\{x, x\} = \{x\}$ is not of size 2.

  The following is not a simple graph: 

.
- There are no parallel edges because $\{\{x, y\}, \{x, y\}\} = \{\{x, y\}\}$.

  The following is not a simple graph: 

.
  Graphs which allow parallel edges or loops are called *multi-graphs*.
- A graph may have cycles.

  Definition 4.1.3 (*tree*)

  A *tree* is a connected acyclic simple graph.

  A collection of trees which share no vertex form a forest. More precisely, contrasted with a DAG (directed acyclic graph):

  Definition 4.1.4 (*forest*)

  A *forest* is an acyclic simple graph.
- Every edge has exactly two vertices—contrasted with a *hypergraph* whose edges may have one or more vertices.

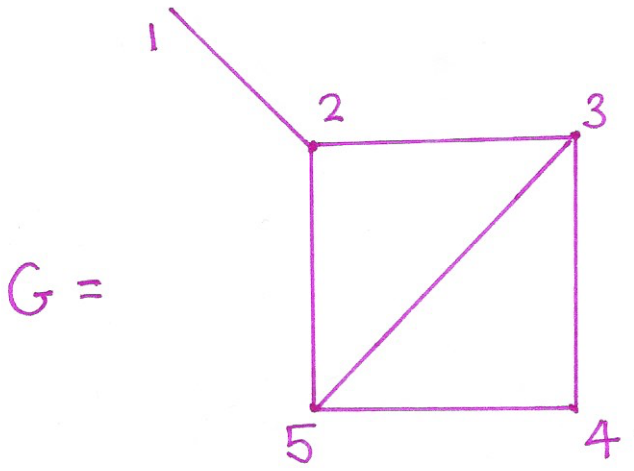
The following terminology is a bit confusing.

Definition 4.1.5 (*Adjacent vertices, and incident edges*)

- If $\{v_1, v_2\}$ is an edge of a graph, we say that $\{v_1, v_2\}$ *joins* or *connects* the vertices v_1 and v_2, and that vertices v_1 and v_2 are *adjacent* (to each other).
- The edge $\{v_1, v_2\}$ is *incident* to vertices v_1 and to v_2.
- Two edges incident to the same vertex are *incident* (to each other).
- A vertex incident to no edge is called *isolated*.

4.2 Adjacency List

The so-called *adjacency list* requires $\Theta(|V| + |E|)$ space, and is efficient when $|E| << |V|^2$.



Let's look again at .

Start with an array indexed by vertex. The value associated with index i is a list of the vertices connected to v_i.

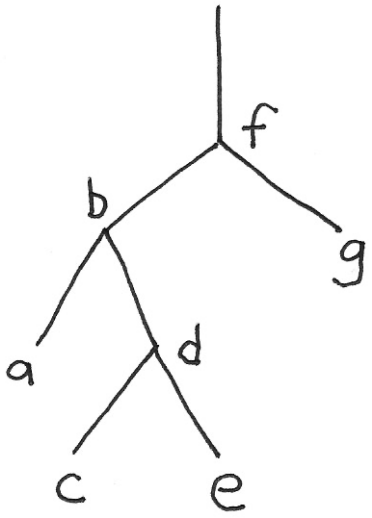
$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{3, 5\}$$
$$A[5] = \{2, 3, 4\}$$

If the graph is un-directed, then whenever $i \in A[j]$, then also $j \in A[i]$. To determine whether v_i connects to v_j we must do a membership check, $i \in V[j]$ or $j \in V[i]$. If the vertices are integers from 0 to $n-1$ or from 1 to n, then A can be a simple array. If not you may have to apply some mapping strategy such as:

- renumbering the vertices
- mapping of the type of vertex to unsigned integer. e.g., String $\rightarrow$ Unsigned.
- hash table

4.3 Depth First Search

To understand DFS, first think of a walk over a binary tree without loops and without diamonds.



Algorithm 1: DFS-binary-tree(v,f)

Input : v vertex of a binary tree
Input : f function to apply to each vertex

```
1 if v.left then
2 |  DFS-binary-tree(v.left,f)
3 f(v)
4 if v.right then
5 |  DFS-binary-tree(v.right,f)
```

We'd like to extend this concept to an arbitrary graph, which:

- may not have a designated root,
- maybe has loops, or
- maybe has multiple edges (> 2) per vertex.

We first look at `DFS-visit` which is almost what we want but has some deficiencies which we will note.

Algorithm 2: DFS-visit(A,v,f,parent)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices
Input : v a starting vertex, $v \in V$
Input : f a function to be applied to each vertex
Input : *parent* maps each vertex to a *parent* vertex or None, $V \to V \cup \{None\}$

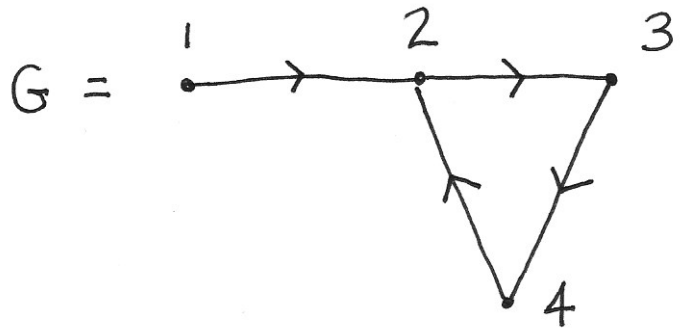
```
1 f(v)
2 foreach neighbor ∈ A[v] do
3 |   if not parent[neighbor] then
4 |   |   parent[neighbor] ← v
5 |   |   DFS-visit(A,neighbor,f,parent)
```

As an optimization and abuse of semantics, the `parent[]` array serves both as the back-pointer from a vertex to the vertex we arrived from, and also as a Boolean to indicate whether the vertex has already been visited, thus avoiding infinite loops. But we must set `parent[s]` to `None` before calling `DFS-visit(...s...)` from external.

What is `None`? It depends on the programming language you are using? Does your programming language support heterogeneous arrays? Can you set `parent[3]="None"` and `parent[4] = 3` ? If so, you can use the string `"None"`. Otherwise you may have to use `0` for `None` and be careful that your vertices are numbered starting at `1`. Or perhaps use `-1` for `None`.

However, be careful that you can distinguish `parent[i]` having not yet been assigned, vs `parent[i]` having been assigned `None`.

If the graph is not connected, `DFS-visit` will fail to visit some vertices. Also if the graph is connected but directed, there is a chance of missing some vertices.



In this graph `DFS-visit(...1...)` would visit every vertex, but `DFS-visit(...3...)` would never visit vertex 1. We need an outer loop, and we can make the code cleaner converting `DFS-visit` to a local function.

Algorithm 3: DFS(A,f)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices
Input : f a function to be applied to each vertex

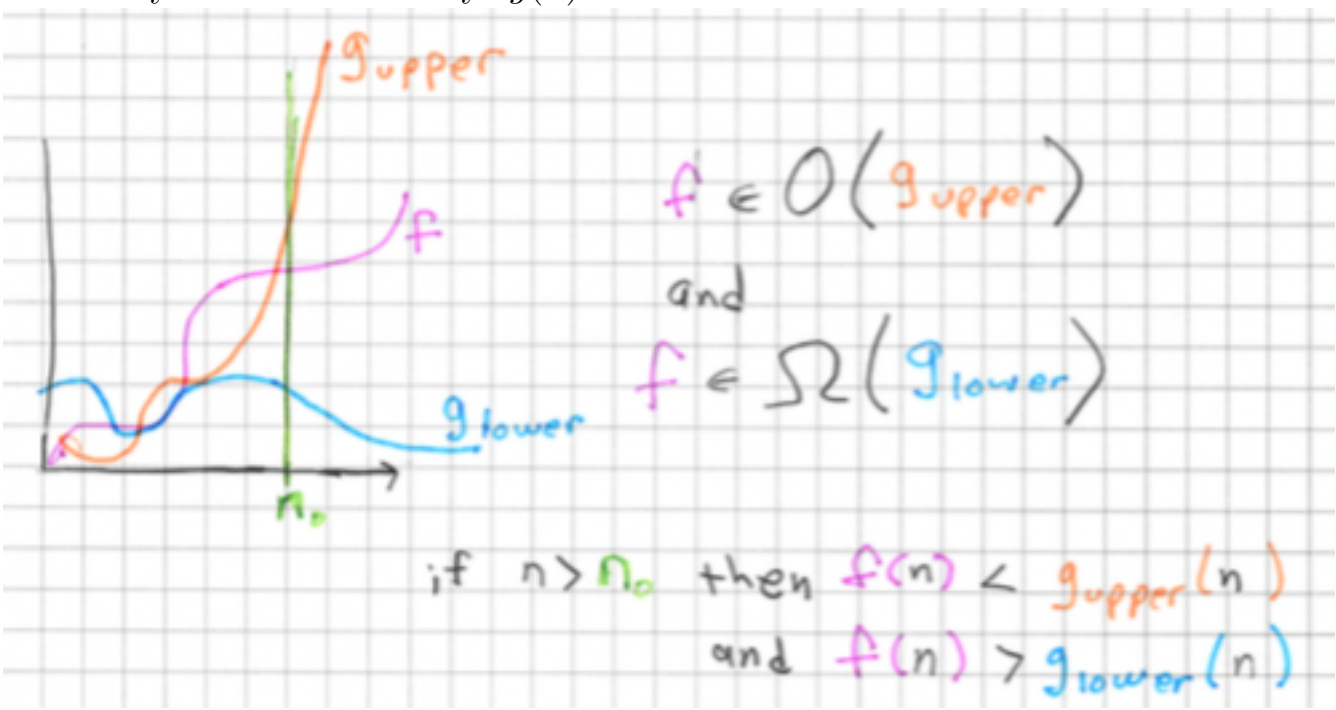
```
1  parent ← ∅ /* empty map                                   */
2  local function DFS-visit(v):
3  |   f(v)
4  |   foreach n ∈ A[v] do
5  |   |   if not parent[n] then
6  |   |   |   parent[n] ← v
7  |   |   |   DFS-visit(n)
   /* main loop of DFS                                      */
8  foreach s ∈ A do
   |   /* indices/keys of A                                 */
9  |   if not parent[s] then
   |   |   /* if s has not yet been visited                 */
10 |   |   parent[s] ← None
11 |   |   DFS-visit(s)
```

To analyze the complexity of `DFS` we know that `DFS-visit` is called exactly once per vertex, and for vertex v_i we perform $O(1) + O(1) \times degree(v_i)$ amount of time assuming $f(i)$ is constant time.

$$
\begin{aligned}
Complexity &= \sum_{i=v_1}^{v_n} O(1) + O(1) \times degree(v_i) \\
&= |V| \times O(1) + O(1) \times \sum_{i=v_1}^{v_n} degree(v_i) \\
&= |V| \times O(1) + O(1) \times 2 \times |E| \\
&= O(|V| + |E|)
\end{aligned}
$$

4.4 Big-O and Big-Ω Notation

Recall that if $f \in O(g)$, then $f(n)$ is eventually bounded above by $cg(n)$ (a constant multiple of $g(n)$ for $n >> 0$, and if $f \in \Omega(g)$, then $f(n)$ is eventually bounded below by $cg(n)$ for $n >> 0$.



4.5 Topological Sort

Topological sort is a way to order a set which is constrained by pairwise orderings. In terms of a graph, topological sort is a way of selecting an order of the vertices of an directed graph such that if (a, b) is an edge of the directed graph, then a precedes b in the sequence of ordered vertices. The problem is normally stated as a set of constraints, and you are asked to produce an ordering which does not violate any of the constraints.

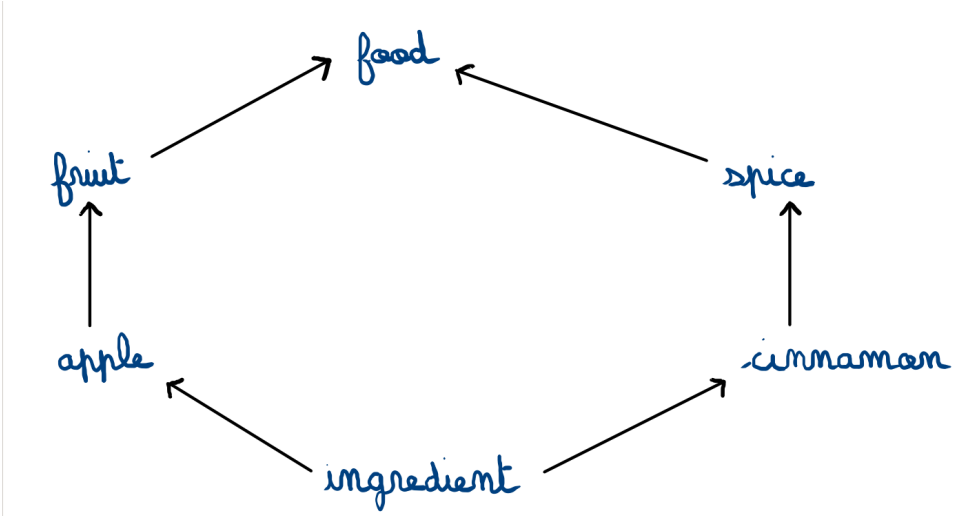
Here is an example from the Common Lisp (CL) programming language. CL provides multiple inheritance. A class is defined by `defclass`; the first argument names to class to be defined, and the second argument lists the direct superclasses.

```
(defclass food () ...)
(defclass fruit (food) ...)
(defclass spice (food) ...)
(defclass apple (fruit) ...)
(defclass cinnamon (spice) ...)
(defclass ingredient (apple cinnamon) ...)
```

In CL, classes are modeled as sets, and subclasses as subsets. If `fruit` is a subclass of `food`, as indicated in the above definition, we write $fruit \subset food$. This leads to the following subset relations.

$$\begin{aligned} ingredient &\subset apple \\ ingredient &\subset cinnamon \\ apple &\subset fruit \\ cinnamon &\subset spice \\ fruit &\subset food \\ spice &\subset food \end{aligned}$$

This leads to a class graph as follows.

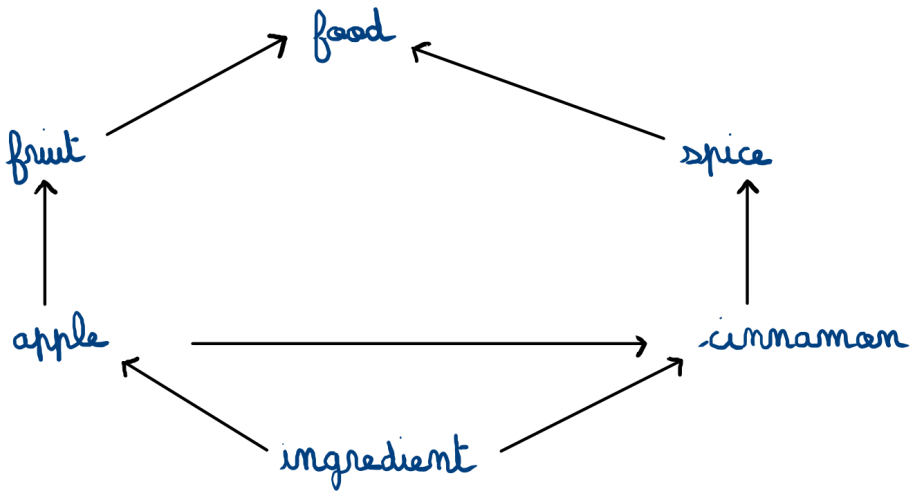


We would like to linearize these classes, respecting all the subclss relations. There are six such orders.

$$
\begin{aligned}
&ingredient \to apple \to cinnamon \to fruit \to spice \to food \\
&ingredient \to apple \to cinnamon \to spice \to fruit \to food \\
&ingredient \to apple \to fruit \to cinnamon \to space \to food \\
&ingredient \to cinnamon \to apple \to fruit \to spice \to food \\
&ingredient \to cinnamon \to apple \to spice \to fruit \to food \\
&ingredient \to cinnamon \to spice \to apple \to fruit \to food
\end{aligned}
$$

It is important when compiling such a program that the CPL (class precedence list) be deterministic, otherwise a running program may have unpredictable behavior. For this reason CL defines additional constraints on the CPL as well as a so-called *tie-breaker* function.

Since the class definition of `ingredient` above mentions `apple` before `cinnamon`, CL requires that `apple` appear before `cinnamon` in the CPL. This adds an additional constraint in the graph.



However, even with this additional constraint, there are still leaves 3 possible orderings.

$$ingredient \to apple \to cinnamon \to fruit \to spice \to food$$
$$ingredient \to apple \to cinnamon \to spice \to fruit \to food$$
$$ingredient \to apple \to fruit \to cinnamon \to spice \to food$$

To alleviate this ambiguity, CL requires that whenever there is an ambiguity, such as $apple \to cinnamon$ or $apple \to fruit$, we must choose the element whose subclass is *farthest to the right* in the list computed so far. So when selecting between `cinnamon` and `fruit` we ask which of these two has a direct subclass farthest to the right in the list computed so far. The list computed so far is $ingredient \to apple$. In this case $apple \subset fruit$ and $ingredient \subset cinnamon$. As `fruit` is further right than `ingredient`, we select $apple \to fruit$, thus leaving only one possibility.

$$ingredient \to apple \to fruit \to cinnamon \to spice \to food$$

4.5.1 Kahn Algorithm (1962)

The following is the Kahn algorithm from wikipedia
`https://en.wikipedia.org/wiki/Topological_sorting`

Algorithm 4: Topological Sort (Kahn)

```
1  L → ∅ list that will contain the sorted elements;
2  S → Set of all nodes with no incoming edge ;
3  while S ≠ ∅ do
4  |   remove a node n from S ;
5  |   concatenate n to tail of L ;
6  |   foreach node m with an edge e from n to m do
7  |   |   remove edge e from the graph;
8  |   |   if m has no other incoming edges then
9  |   |   |   insert m into S ;
10 if graph has edges then
11 |   return error (graph has at least one cycle) ;
12 else
13 |   return L (a topologically sorted order)
```

The original Kahn algorithm did not explicitly mention a user defined tie-breaker function.

In terms of the CL Topological sort algorithm for calculating a deterministic CPL, the Kahn algorithm proceeds as follows.

List the constraints:

$$\begin{aligned} ingredient &\rightarrow apple \\ ingredient &\rightarrow cinnamon \\ apple &\rightarrow cinnamon \\ apple &\rightarrow fruit \\ cinnamon &\rightarrow spice \\ fruit &\rightarrow food \\ spice &\rightarrow food \end{aligned}$$

Start with the class being defined. This class is unconstrained, nothing is inheriting from it by definition, because we are in the process of defining it. I.e., initialize

$$CPL = ingredient$$

and mark all constraints $ingredient \rightarrow x$ as satisfied.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
$apple \rightarrow cinnamon$	
$apple \rightarrow fruit$	
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Now find the set of unconstrained class(es). The only such class is `apple`, so add it to the *end* of the CPL.

$$CPL = ingredient \rightarrow apple$$

And remove all constraints on `apple`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Now find the set of unconstrained classes. They are $\{cinnamon, fruit\}$. Kahn's original algorithm didn't consider tie breaking. So by that original algorithm, you could take either `cinnamon` or `fruit` as you prefer.

However, we will apply the tie breaker rule. Look at the satisfied constraint arrows leading to `cinnamon` and to `fruit`, but only those corresponding to subclass relationships: $ingredient \rightarrow cinnamon$ and $apple \rightarrow fruit$. Which one of these originates from the class furthest to the right in the thus far computed list $ingredient \rightarrow apple$? The answer is $apple \rightarrow fruit$, so we take `fruit` rather than `ingredient`, and eliminate constraints on `fruit`.

$$CPL = ingredient \rightarrow apple \rightarrow fruit$$

And remove all constraints on `fruit`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

Now, the only unconstrained class is `cinnamon`, so we update the CPL

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon$$

And remove all constraints on `cinnamon`.

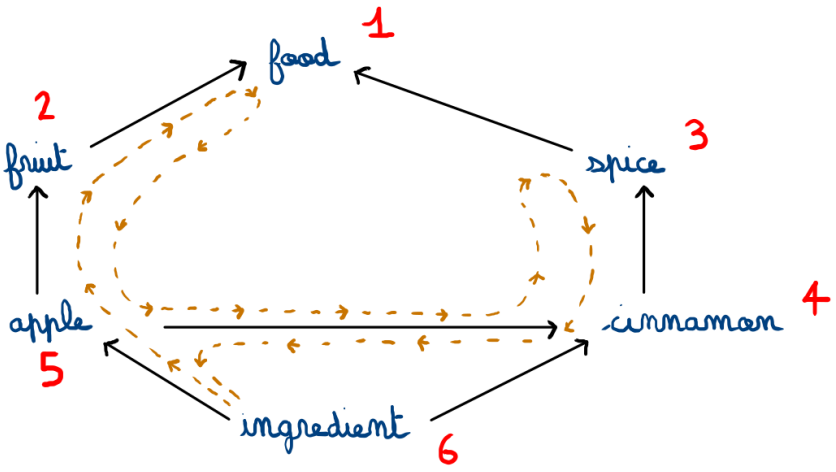
~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
~~$cinnamon \rightarrow spice$~~	SATISFIED
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

We are now only left with `spice` and `food`, so we append them to the CPL.

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

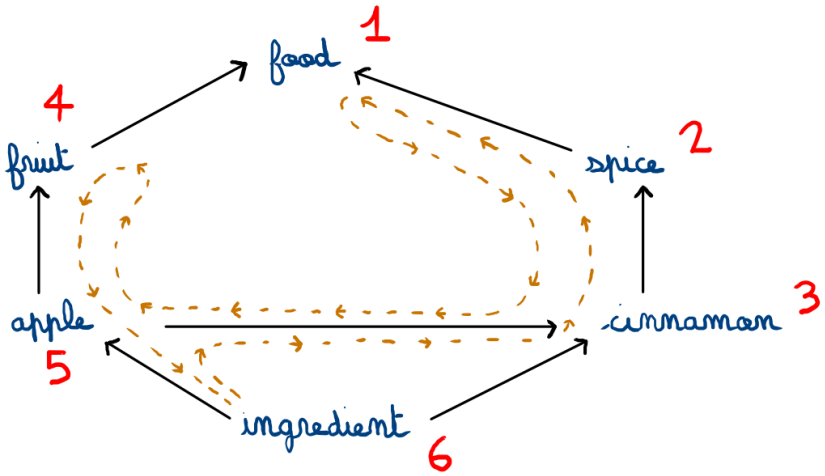
4.5.2 Tarjan Algorithm (1976)

To obtain a topologically sorted list of vertices of a graph, we may perform a DFS and annotate the vertices after visiting all the neighbors. Depending on the semantics of the arrow directions, either the resulting order or its reverse is the topological order. Since the neighbors are unordered, multiple topological orders are possible.



Now reverse the numbered vertices

6 *ingredient* → 5 *apple* → 4 *fruit* → 3 *cinnamon* → 2 *spice* → 1 *food*



6 *ingredient* → 5 *apple* → 4 *cinnamon* → 3 *spice* → 2 *fruit* → 1 *food*

4.5.3 Kahn more general than Tarjan

Consider the directed graph $G = (V, E)$ whose vertices are

$$V = \{a, b, c, d, e, f\}$$

There are many ways to topological sort the vertices of this graph subject to the constraints $\{(a, b), (a, c), (b, c), (b, d), (d, f), (c, e), (e, f)\}$. I.e., a precedes b, a precedes c, b precedes c etc.

- a b c d e f
- a b c e d f
- a b d c e f

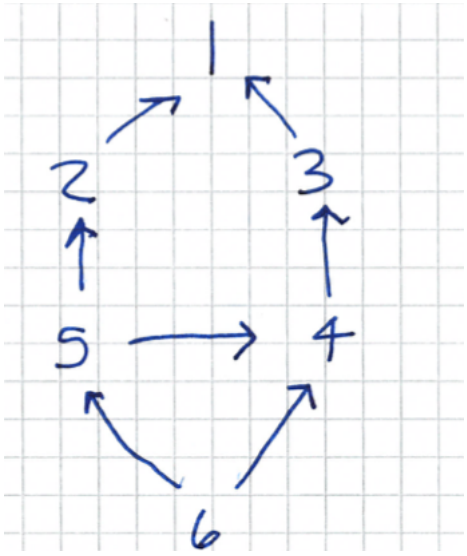
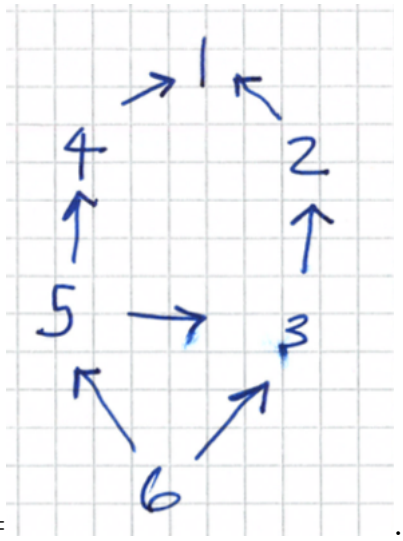
But we would like to find the order of the vertices which comes first in lexigraphical order. Using the Kahn algorithm we can use a tie-breaker rule that when given a choice of multiple vertices to select, we choose the one that comes first in alphabetical order. This gives

- a b c d e f

However, this order is not obtainable from the Tarjan algorithm because in every DFS of the graph, c and e are adjacent.

4.5.4 Topological sort as graph isomorphism

If we look at the two graphs in Section 4.5.2, the act of numbering the nodes is equivalent to finding an isomorphic graph.

$G =$  and $H =$  .

If we examine the adjacency matrix of G and H we see that they are both lower triangular.

$$A_G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

$$A_H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

We see that another way to calculate a topological sort is transform the adjacency matrix into a lower triangular matrix by swapping rows and columns.

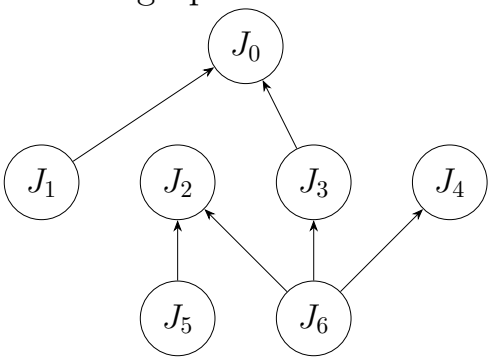
4.6 Job Scheduling

Given a set of interdependent jobs you wish to run. How can we calculate which order to run the jobs so that all dependencies are met? Suppose that jobs $\{J_0, J_1, J_2, J_3, J_4, J_5, J_6\}$ need to run. But some are dependent on others. We cannot start J_0 until both J_1 and J_3 are finished. I.e., J_1 and J_3 depend on J_0.

Suppose $a \to \{b_1, b_2, \ldots b_n\}$ means a cannot start before $b_1 \ldots b_n$ all finish, or a depends on b. Consider a set of constraints as follows:

$$\begin{aligned} J_1 &\to \{J_0\} \\ J_3 &\to \{J_0\} \\ J_6 &\to \{J_2, J_3, J_4\} \\ J_5 &\to \{J_2\} \end{aligned}$$

The graph looks as follows:

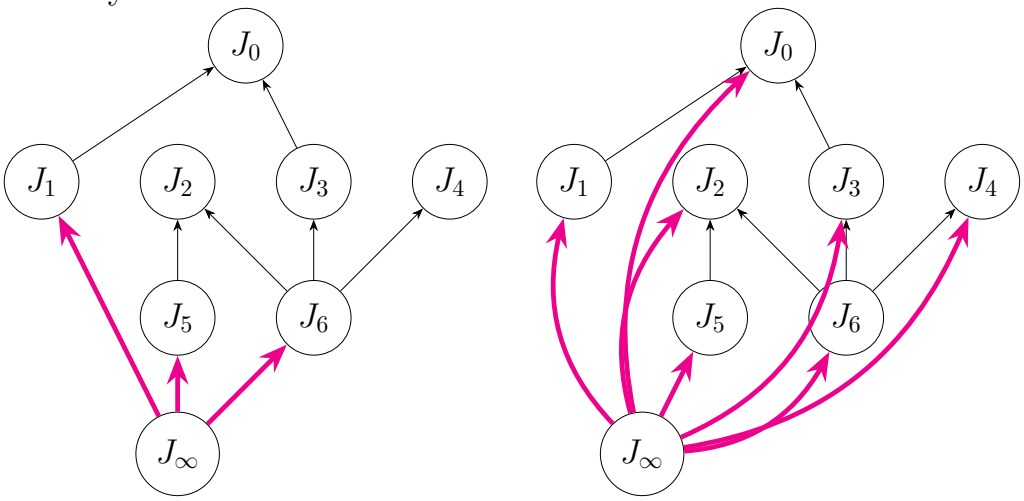


4.6.1 Scheduling using DFS

We may use DFS to label the vertices of this graph into a topological order, obeying the constraints.

If the jobs must execute sequentially, i.e., we cannot run multiple jobs at once, then any valid order is just as good as any other. So any topologically sorted order will suffice.

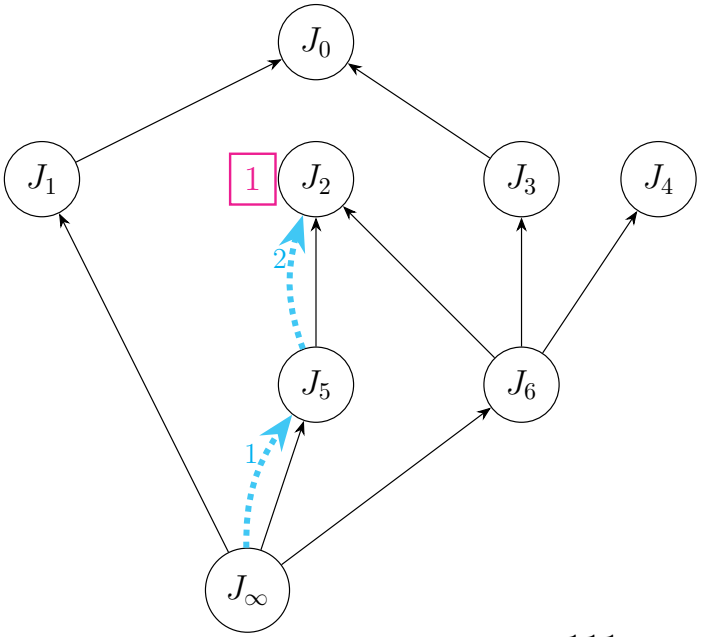
Since we have multiple starting (unconstrained) vertices $\{J_1, J_5, J_6\}$ we don't know where to start the DFS. There are two strategies which can be helpful. We may create an *artificial* vertex, J_∞ leading to the unconstrained vertices $\{J_1, J_5, J_6\}$, or we may instead connect J_∞ to all the vertices of the graph. The latter strategy obviates the need to actually identify the unconstrained vertices.



We can then perform the simple DFS starting at vertex J_∞. Labeling each node after all its neighbors have been visited.

The following is a step-by-step walk through of the DFS.

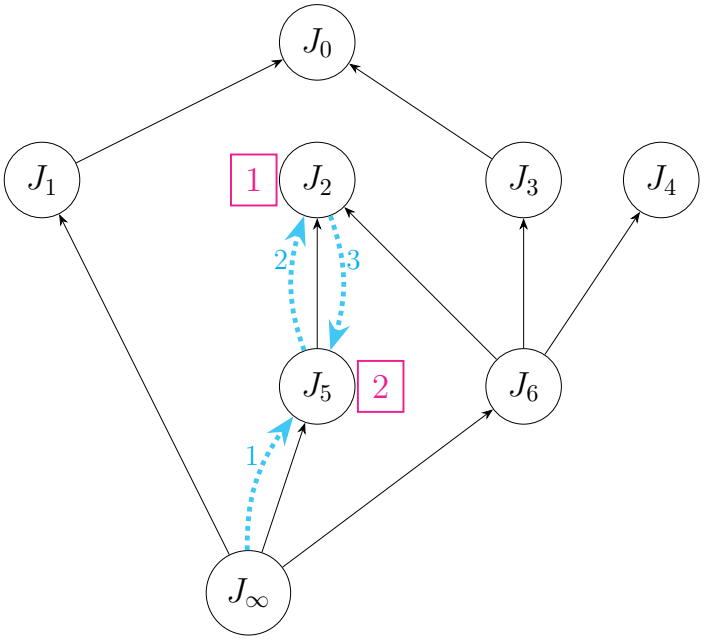
Start at vertex J_∞



Starting at vertex J_∞ we see that J_∞ has three neighbors: J_1, J_5, and J_6. While the Kahn algorithm (Section 4.5.1) allows a tie-breaking rule to eliminate ambiguity, the Targan (DFS) algorithm typically does not. Lacking a deterministic decision algorithm, we simply make an arbitrary choice. For the purpose of exposition, we choose to recur on J_5. At

each step of the recursion we have to as: Are there any remaining *unvisited* neighbors. J_5 has only one neighbor, J_2, which is in fact not-yet visited, so we recur to J_2. J_2 has no neighbors (in general no remaining not-yet-visited neighbors), so we mark the vertex J_2 with a counter, starting at $order = \boxed{1}$.

Backtrack to J_5



Since J_2 is finished and has been marked with $order = \boxed{1}$, we backtrack to J_5, as labeled 3 in the figure, and ask J_5 has any remaining not-yet-visited neighbors?

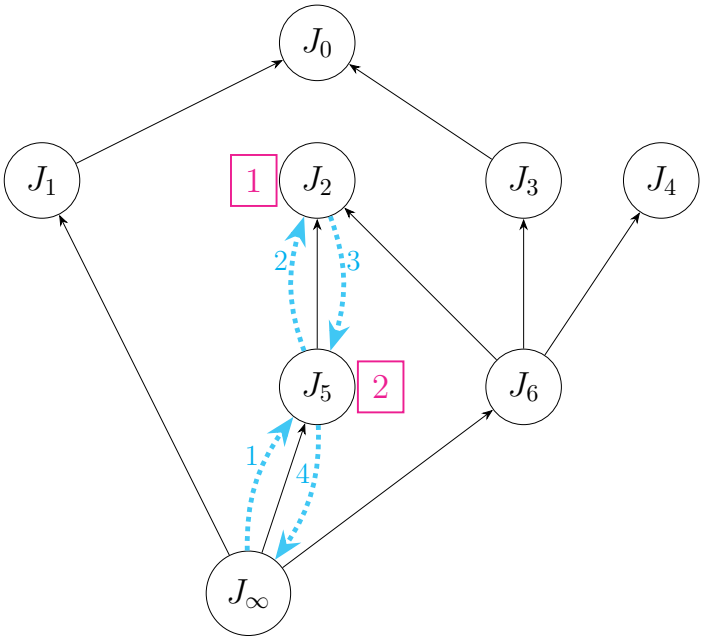
The answer is no, so we mark J_5 with $order = \boxed{2}$.

At this point the as-yet-incomplete topological order is

$$\{J_2, J_5, \ldots\}.$$

We will complete this sorting in the upcoming steps.

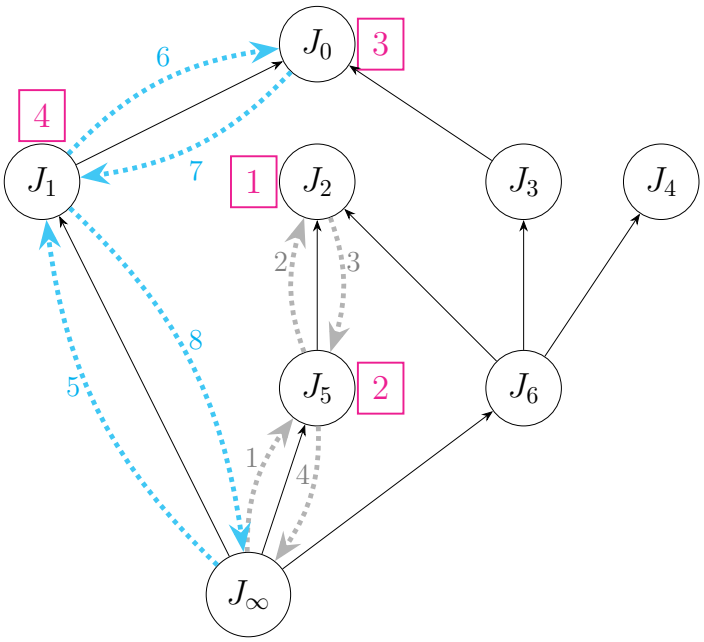
Backtrack to J_∞



Since J_5 has no remaining not-yet-visited neighbors, we backtrack to J_∞, as labeled 4 in the figure.

Does, J_∞ have remaining unvisited neighbors? Yes, so we don't assign any order label yet.

Advance to J_1



J_∞ has are two unvisited neighbors: J_1 and J_6, so we arbitrarily choose J_1.

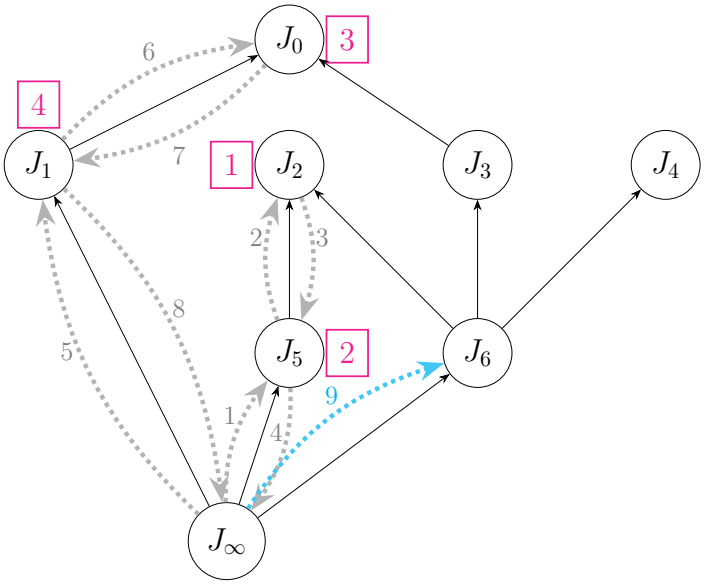
J_1 has one not-yet-visited neighbor, namely J_0, which has no neighbors at all. So we label J_0 with $order = \boxed{3}$, and backtrack to J_1. J_1 at this time has no unvisited neighbors, so we label it $order = \boxed{4}$.

J_1 being finished allows us to backtrack to J_∞ for further processing.

This excursion is labeled 5, 6, 7, 8 in the figure.

At this point the as-yet-incomplete topological order is $\{J_2, J_5, J_0, J_1, \ldots\}$.

Advance to J_6



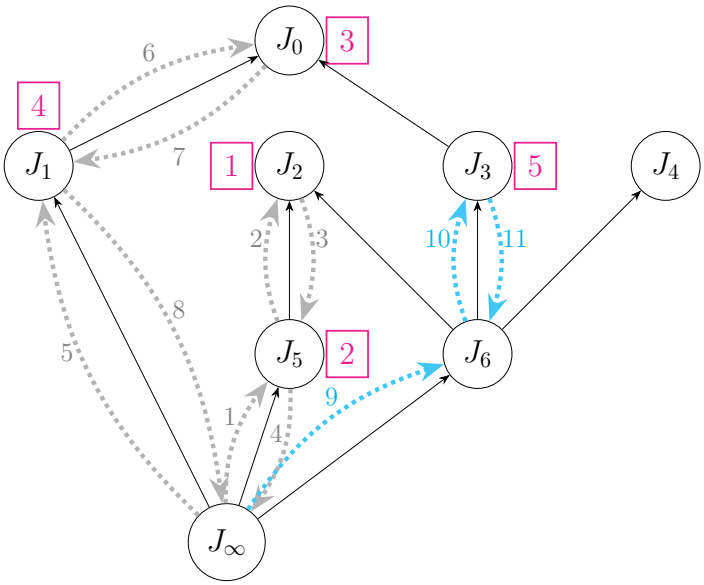
There remains only one unvisited neighbor of J_∞ which is J_6, so we proceed to as labeled 9 in the figure.

J_6 has three neighbors: J_2, J_3, and J_4, but J_2 has already been visited so we arbitrarily choose between J_3 and J_4.

No vertex is labeled in this step, because we did not *finish* any vertex.

At this point the as-yet-incomplete topological order remains $\{J_2, J_5, J_0, J_1, \ldots\}$, because no vertex with labeled with an *order*.

Advance to J_3

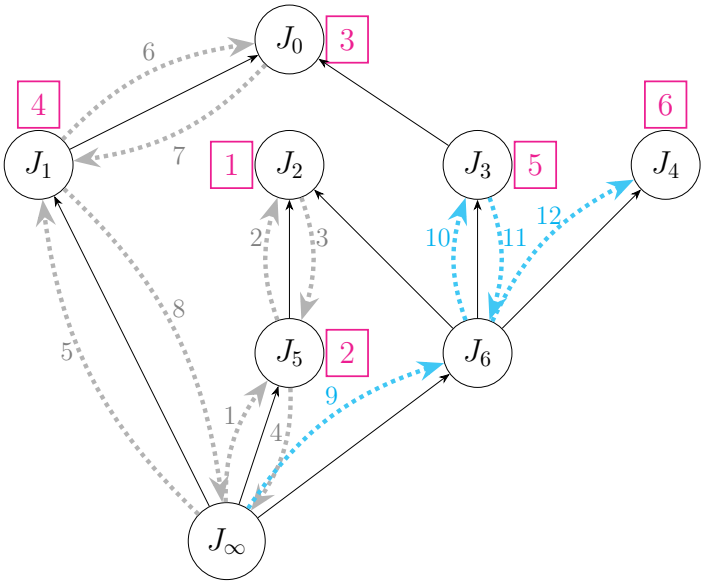


To advance, we arbitrarily choose J_3, from the possible choices of J_3 and J_6. J_3 has one neighbor, J_0, which has already been visited. Since all the neighbors of J_3 have now been visited we label J_3 with $order = \boxed{5}$.

The topological order (in work) is now $\{J_2, J_5, J_0, J_1, J_3, \ldots\}$.

Having finished with J_3 we backtrack to J_6 and continue examining its neighbors.

Advance to J_4

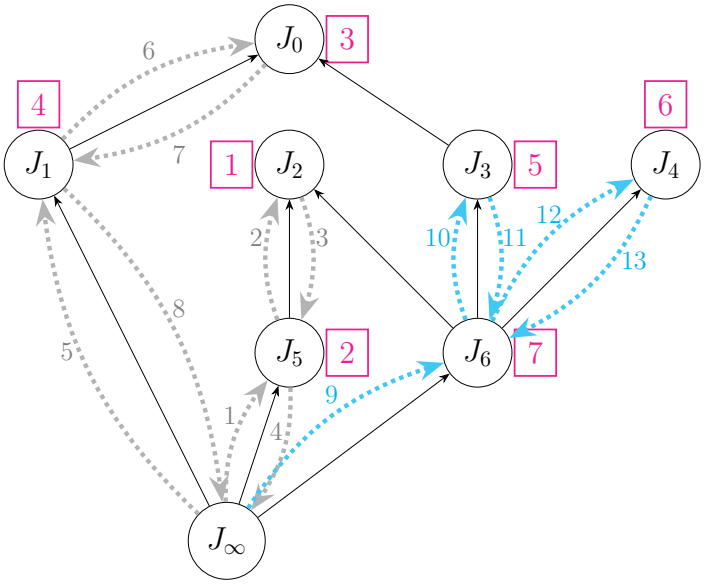


The only remaining not-yet-visited neighbor of J_6 is J_4, so we advance to J_4.

J_4 has no neighbors, and thus no unvisited neighbors, so we label it $order = \boxed{6}$.

Having identified the $order = \boxed{6}$ vertex, we now have a topological order (in work) of $\{J_2, J_5, J_0, J_1, J_3, J_4, \ldots\}$.

Backtrack to J_6

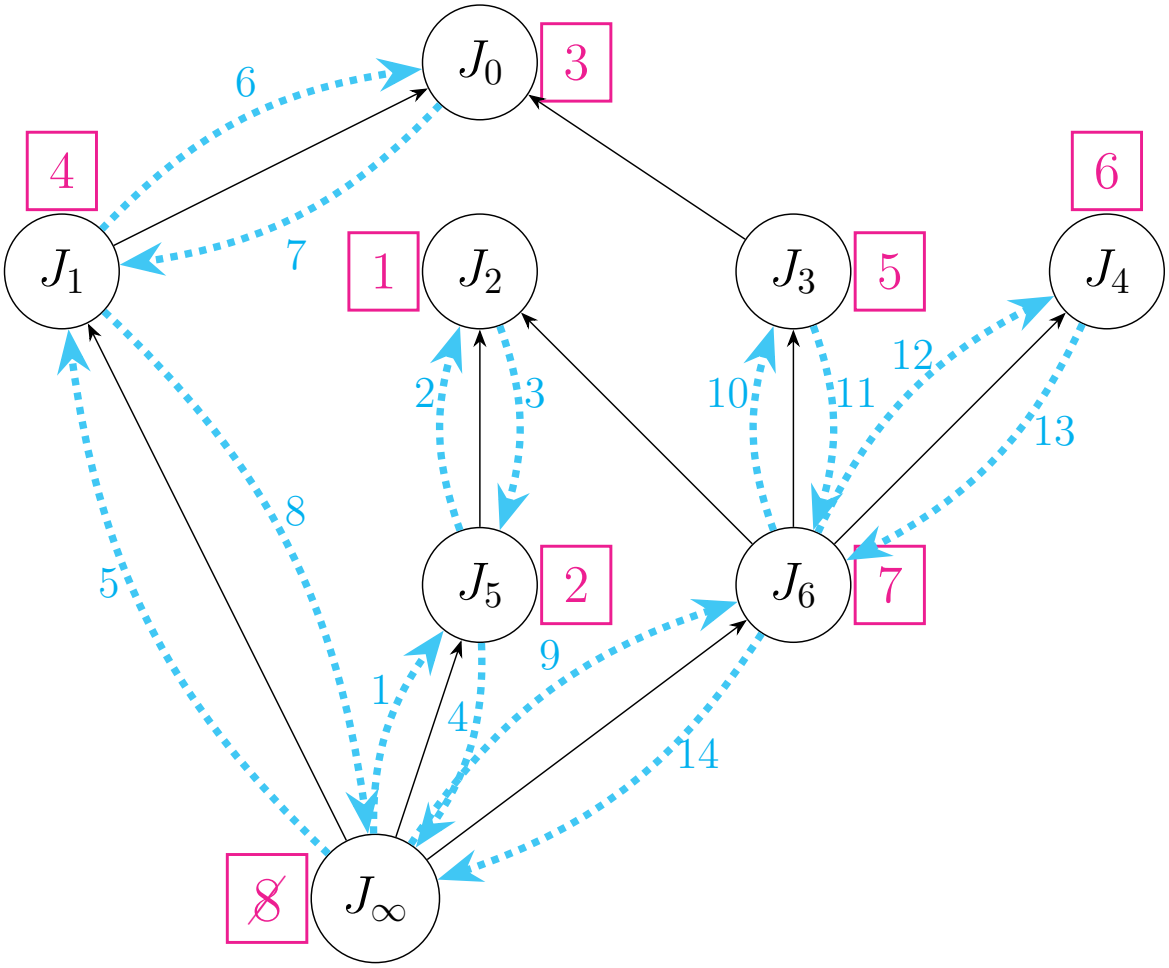


Having finished with J_4, we backtrack to J_6 and find there are no remaining unvisited neighbors. We thus label J_6 with $order = \boxed{7}$.

This brings the topological order to $\{J_2,J_5,J_0,J_1,J_3,J_4,J_6,\}$.

We don't know it yet, but this is the complete topological order, as we will find no additional vertices in the remaining steps of the DFS.

DFS Finished



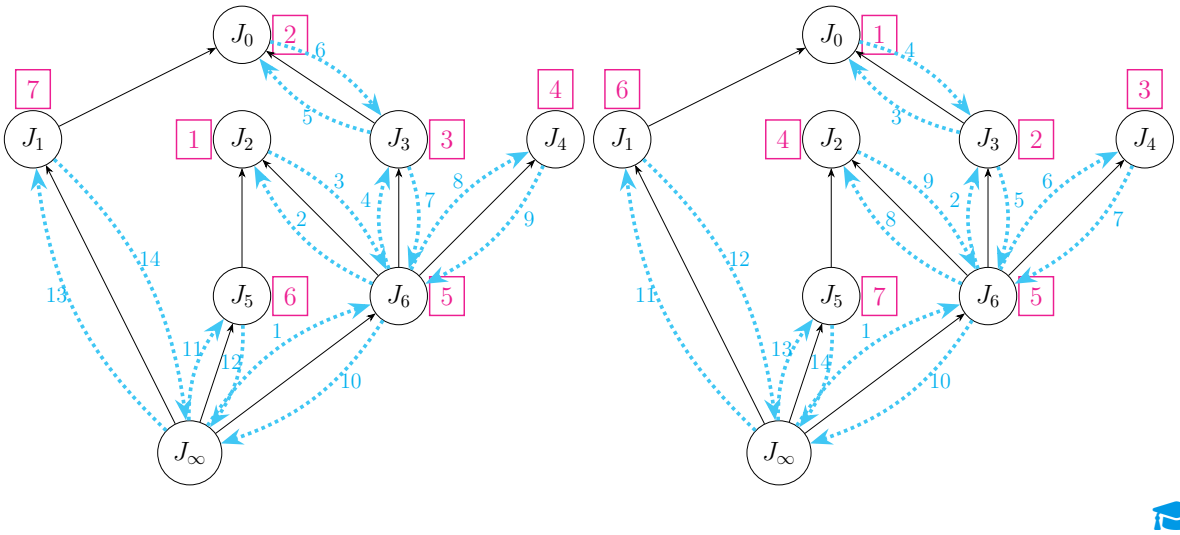
Having finished J_6, we backtrack further to J_∞.

Since there are no remaining unvisited neighbors of J_∞, and J_∞ was the entry vertex of the DFS, the DFS completes.

We could label J_∞ with $order = \boxed{8}$, but that is unnecessary as this was a vertex not in the original list. We simply added this supplementary vertex to make the DFS algorithm simpler—to avoid having unconstrained vertices.

The final topological order is $\{J_2, J_5, J_0, J_1, J_3, J_4, J_6\}$.

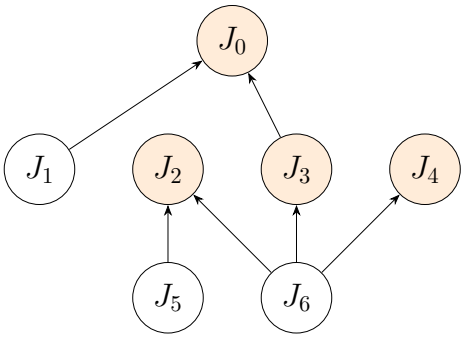
Recall that during the DFS, several times we made an arbitrary choice about which unvisited neighbor to visit next. If we had made different choices, we would have found a different topological order. Many such orders are possible. The following figure shows two such possibilities: $\{J_2, J_0, J_3, J_4, J_6, J_5, J_1\}$, and $\{J_0, J_3, J_4, J_2, J_6, J_1, J_5\}$, respectively.



4.6.2 Job Scheduling with Multiple Threads

Now we revisit the scheduling problem assuming we have two threads to use–we may run two (but no more) jobs simultaneously, as long as we never violate the inter-job dependencies stated earlier. We further assume that we have estimates on how long the jobs will each take, and we can use this prediction to decide which order to visit the neighbors, and thus which order the jobs are launched.

Job No.	**Estimated Time (sec)**
0	3
1	4
2	6
3	2
4	5
5	3
6	10

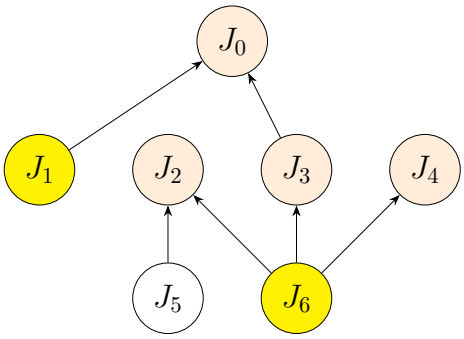


J_0	Blocked
J_1	
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	
J_6	

Whenever we have the choice of 2 or more jobs to start, we will start the one projected to run slowest.

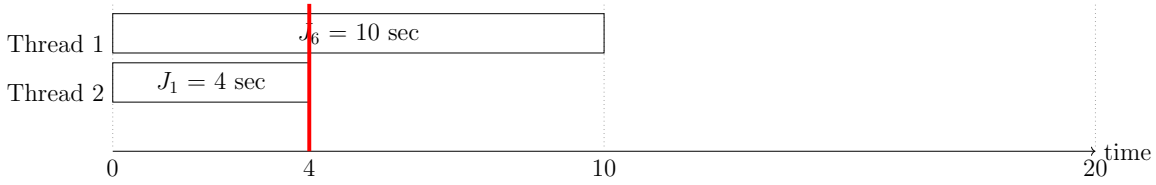
Time $t = 0$

Initially, we can choose among J_1, J_5, and J_6. The slowest two are $J_6, \Delta t = 10$ and $J_1, \Delta t = 4$.

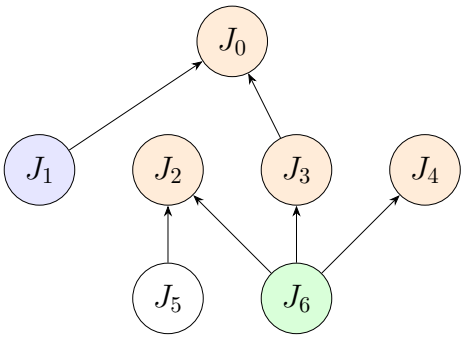


J_0	Blocked
J_1	Started
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	
J_6	Started

Time $t = 4$

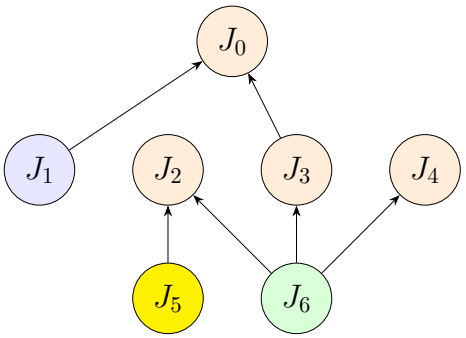


J_1 finishes at $t = 4$, leaving J_6 running and the following graph.



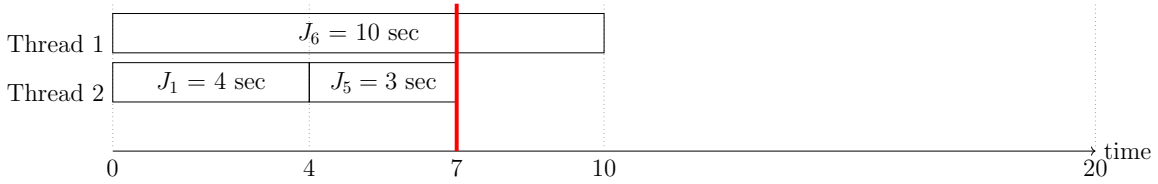
J_0	Blocked
J_1	Finished
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	
J_6	Running

There is only one job which we can start, J_5.

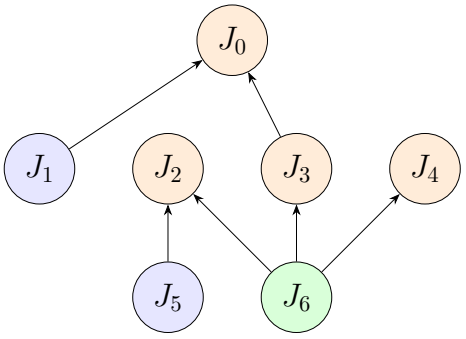


J_0	Blocked
J_1	Finished
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	Started
J_6	Running

Time $t = 7$



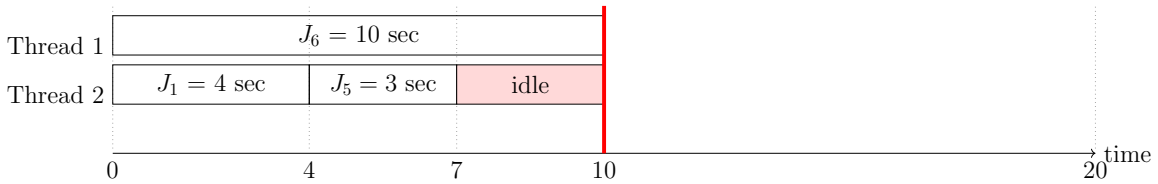
At after $\Delta t = 3$, at time $t = 7$, job J_5 finishes, but J_6 is still running.



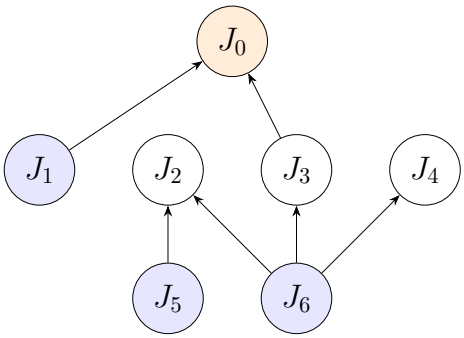
J_0	Blocked
J_1	Finished
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	Finished
J_6	Running

At this point we cannot start any new job. We must wait with thread 2 idle until J_6 finishes at time $t = 10$.

Time $t = 10$



After job J_6 finishes at time $t = 10$ the remaining dependency graph looks like the following.

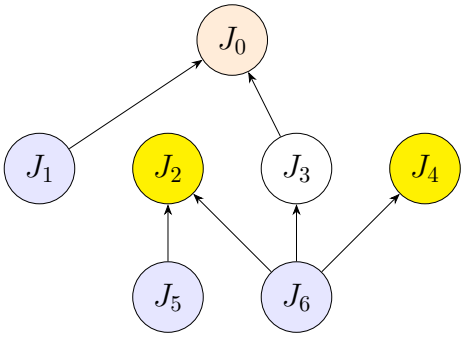


J_0	Blocked
J_1	Finished
J_2	
J_3	
J_4	
J_5	Finished
J_6	Finished

At this point there are two threads free, so we can start two jobs. There are three candidates:

J_2 $\Delta t = 6$
J_3 $\Delta t = 2$
J_4 $\Delta t = 5$

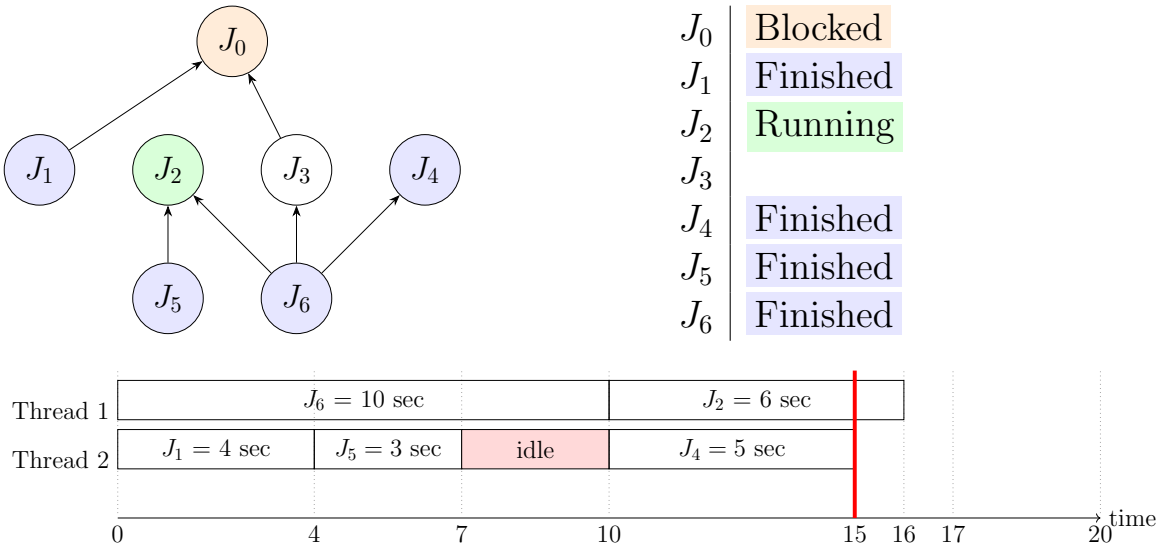
Choosing the slowest two of the three: i.e., J_2 and J_4.



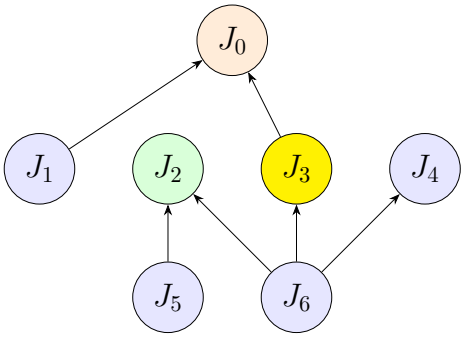
J_0	Blocked
J_1	Finished
J_2	Started
J_3	
J_4	Started
J_5	Finished
J_6	Finished

Time $t = 15$

Job J_4 finishes in $\Delta t = 5$ seconds at time $t = 15$.



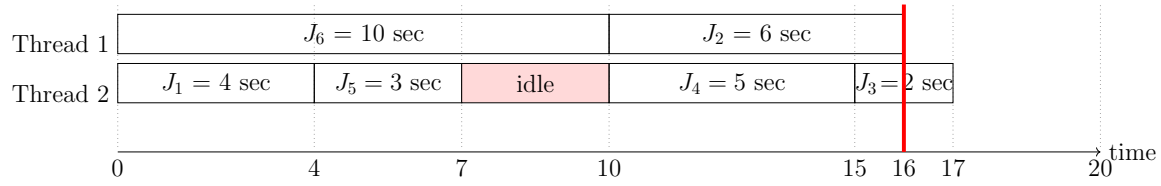
This leaves J_2 running, but J_3 can be launched.
So we start job J_3 at time $t = 15$.



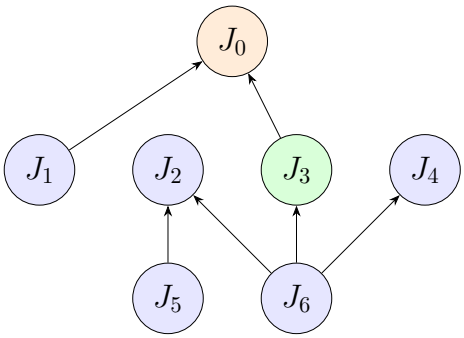
J_0	Blocked
J_1	Finished
J_2	Running
J_3	Started
J_4	Finished
J_5	Finished
J_6	Finished

Time $t = 16$

At time $t = 16$, job J_2 finishes, having run $\Delta t = 6$ seconds.



This leaves the following dependency graph.

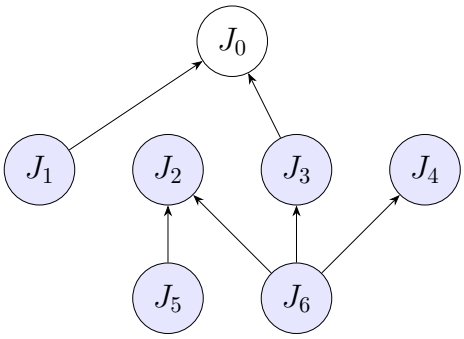


J_0	Blocked
J_1	Finished
J_2	Finished
J_3	Running
J_4	Finished
J_5	Finished
J_6	Finished

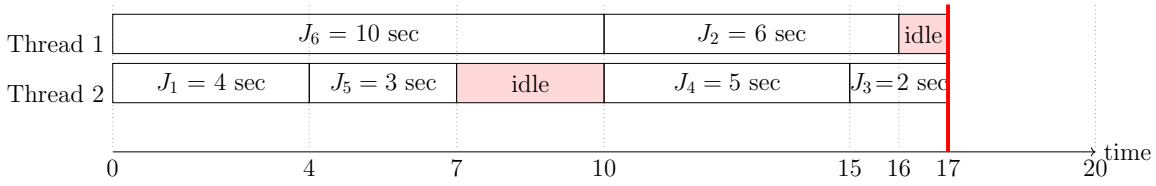
Since job J_3 is still running, we cannot launch any more jobs until J_3 finishes, leaving thread 1 idle for one second until time $t = 17$.

Time $t = 17$

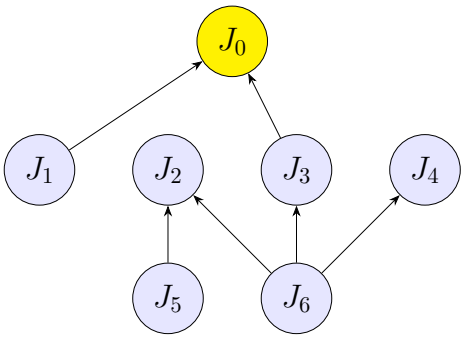
At time $t = 17$, job J_3 finishes.



J_0	
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Finished
J_5	Finished
J_6	Finished



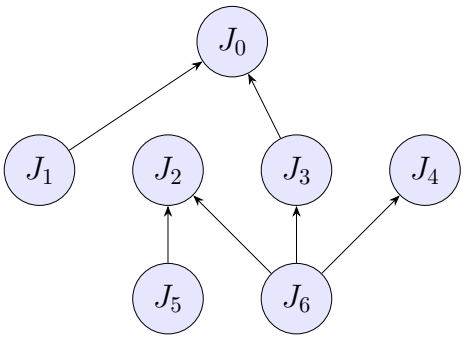
Now, both threads are available, but the only remaining job to run is J_0 which is unblocked. We may launch the final job J_0, leaving one thread idle for $\Delta t = 3$ seconds.



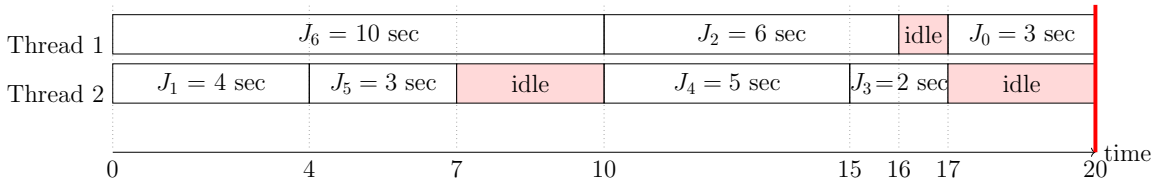
J_0	Started
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Finished
J_5	Finished
J_6	Finished

Time $t = 20$

The total wall time of execution is $\Delta t = 20$ seconds, including 7 seconds of idle time.



J_0	Finished
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Finished
J_5	Finished
J_6	Finished



4.6.3 Job Scheduling with Worst-case First

In this section we rework the example from Section 4.6.2, but whenever we have a choice of job to start we use a different algorithm that was used in Section 4.6.2. In this case we will choose the job which has the worst case completion time when adding the job's predicted time together with each of its children, recursively.

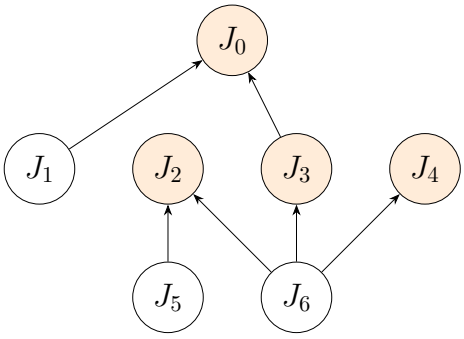
For example, the worst case for J_6 will be

$$\max\{J_6 + J_2, J_6 + J_3 + J_0, J_6 + J_4\} = \max\{16, 15, 15\} = 16\,,$$

and then use the tie-breaker where we start the job with the maximum estimated time. What changes compared to the previous section?

First we revise the time estimates.

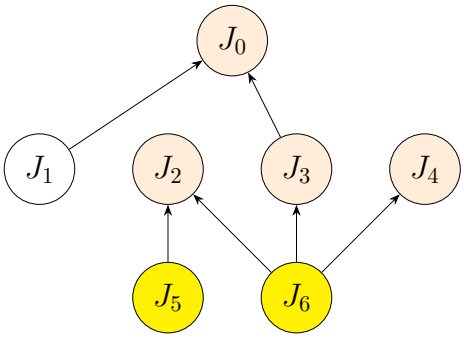
Job No.	**Estimated Time (sec)**	**Δt, Worst Case**
0	3	3
1	4	$4 + 3 = 7$
2	6	6
3	2	$2 + 3 = 5$
4	5	5
5	3	$3 + 6 = 9$
6	10	$10 + \max\{6, 5, 5\} = 16$



J_0	Blocked
J_1	
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	
J_6	

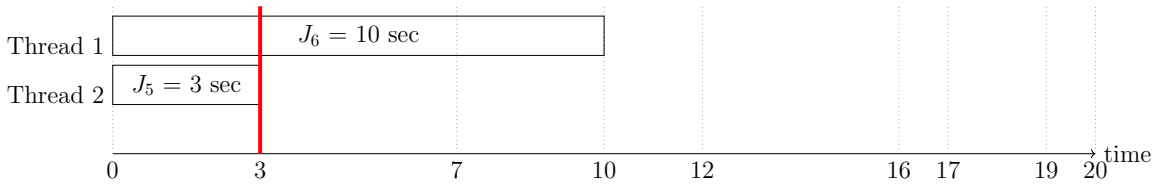
Time $t = 0$

Of the three jobs which are available to start: J_1, J_5, and J_6, the two worst case estimates are $J_6, \Delta t = 16$ and $J_5, \Delta t = 9$.

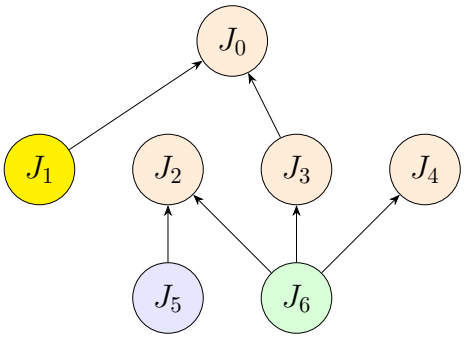


J_0	Blocked
J_1	
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	Started
J_6	Started

Time $t = 3$

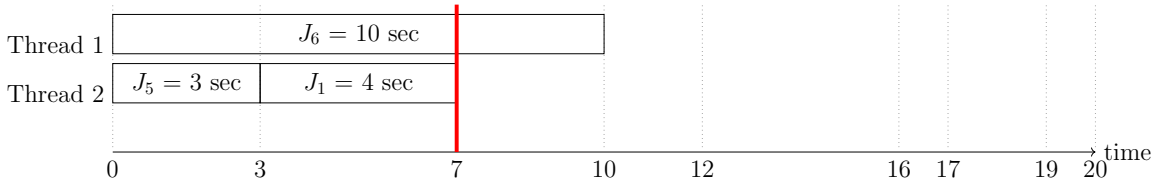


At time $t = 3$, J_5 finishes and thread 2 becomes free. There is only one job which we can start, J_1 which is estimated to run for 4 seconds.

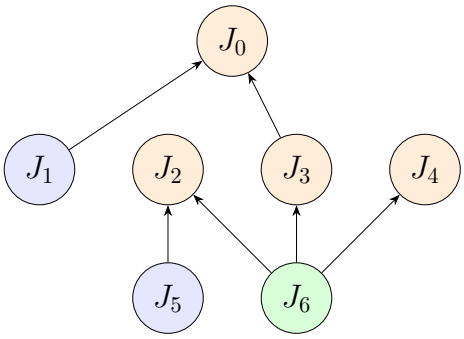


J_0	Blocked
J_1	Started
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	Finished
J_6	Running

Time $t = 7$

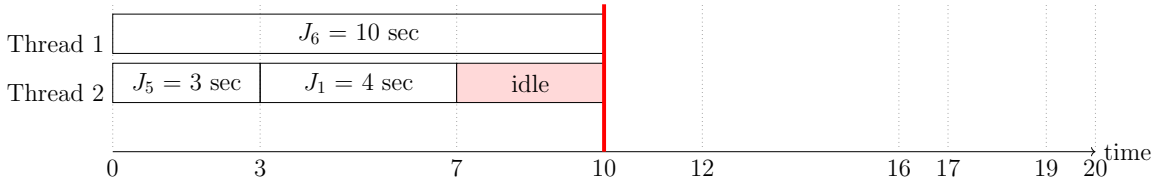


At time $t = 7$ job J_1 finishes, but all the other jobs are blocked. We have no choice but to wait until J_6 finishes. Thread 2 remains idle until that time.

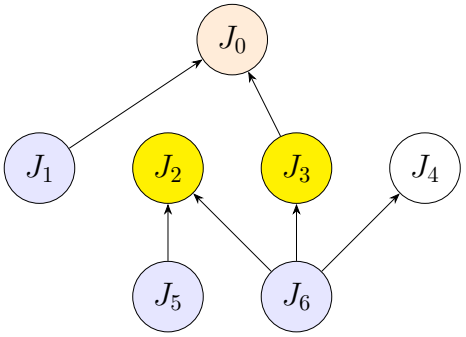


J_0	Blocked
J_1	Finished
J_2	Blocked
J_3	Blocked
J_4	Blocked
J_5	Finished
J_6	Running

Time $t = 10$

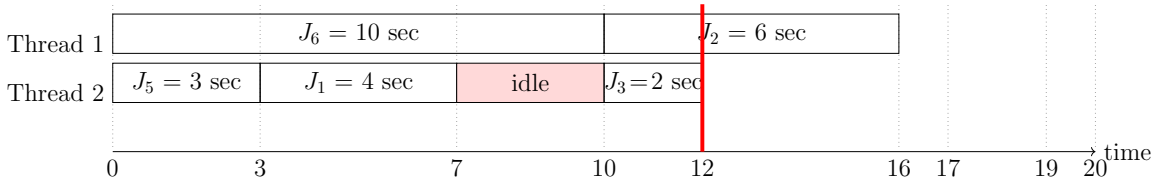


At time $t = 10$, job J_6 finishes, freeing thread 1, and also unblocking jobs J_2, J_3, and J_4. Since two threads are free, we can start two jobs. J_2 has worst case $\Delta t = 6$, while J_3 and J_4 both have worst case $\Delta t = 5$. We start J_2 (which should finish in 6 seconds) and arbitrary choose J_3 (which should finish in 2 seconds).

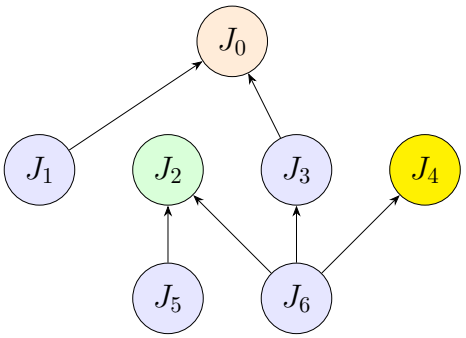


J_0	Blocked
J_1	Finished
J_2	Started
J_3	Started
J_4	
J_5	Finished
J_6	Finished

Time $t = 12$

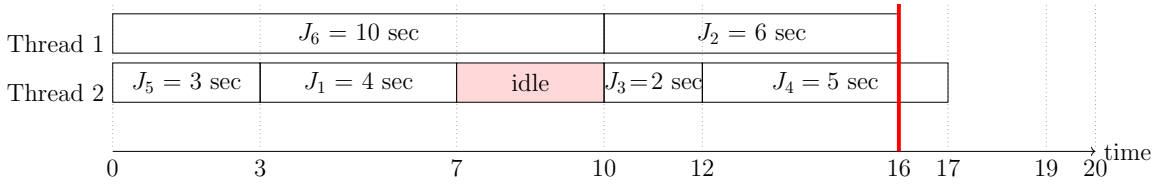


At time $t = 12$ job J_3 finishes, freeing thread 2. Since J_4 is the only job which is not blocked, we start it in thread 2. J_4 should finish in 5 seconds.

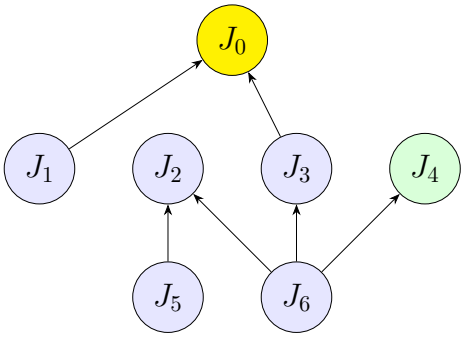


J_0	Blocked
J_1	Finished
J_2	Running
J_3	Finished
J_4	Started
J_5	Finished
J_6	Finished

Time $t = 16$

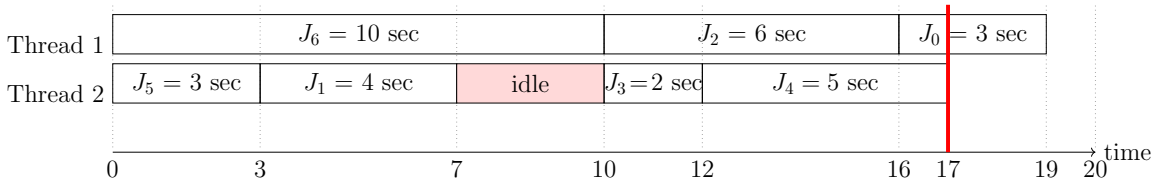


At time $t = 16$ job J_2 finishes, freeing thread 1, and finally unblocking J_0. Since J_0 is the only job which is not blocked, we start it in thread 1. J_0 should finish in 3 seconds.

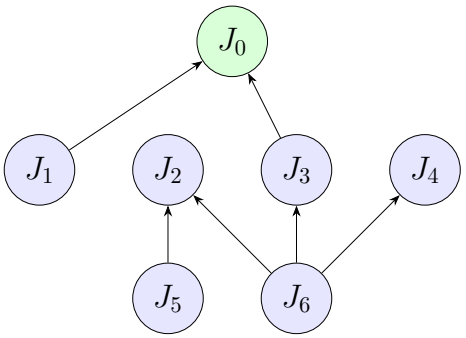


J_0	Started
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Running
J_5	Finished
J_6	Finished

Time $t = 17$

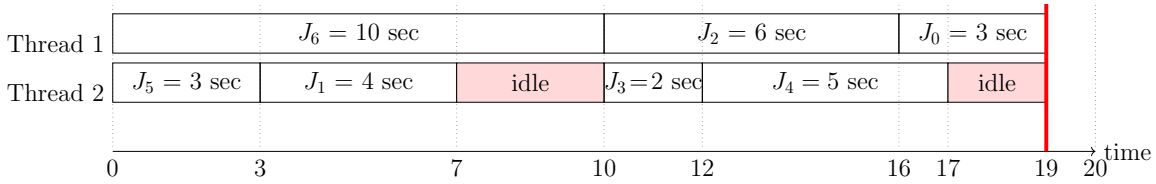


At time $t = 17$, job J_4 finishes, freeing thread 2, but there are no more jobs to start. So thread 2 must remain idle.

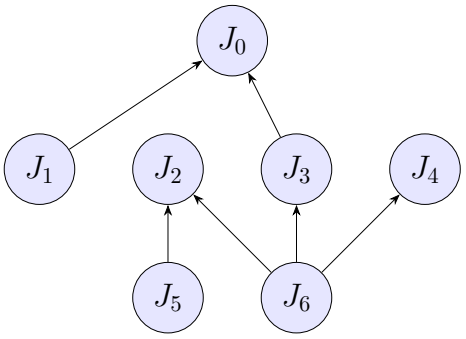


J_0	Running
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Finished
J_5	Finished
J_6	Finished

Time $t = 19$



Finally, at time $t = 19$ job J_0 (the last job) finishes.



J_0	Finished
J_1	Finished
J_2	Finished
J_3	Finished
J_4	Finished
J_5	Finished
J_6	Finished

Strategy	**Wall-time**	**Idle-time**
Fast First	20	7
Slow First	17	5

The total wall-time is 19 seconds, with a total of $3 + 2 = 5$ seconds of thread idle time. Compare this result to the previous result in Section 4.6.2 where the total wall-time was 20 seconds with 7 seconds of idle time.

4.6.4 Exercises for the student

Rework the example using three threads, with any tie breaker you wish or invent a new one.

4.7 Homework

There are two homework assignments `homework/simplegraphs.py` and `homework/topologicalsort.py`; the test cases are found in `tests/simplegraphs_test.py` and `tests/topologicalsort_test.py`. The student should implement the functions which are incomplete in these files.

Chapter 5

Abstract Syntax Trees

Chapter Objectives

- Define AST and ADT; represent expressions as labeled n-ary trees
- Evaluate ASTs recursively using an operator dispatch table
- Apply memoization (`@cache`) to avoid redundant sub-problem re-computation
- Solve the matrix chaining problem by choosing optimal split points
- Model expression types with an OO class hierarchy mirroring the ADT
- Use multiple inheritance for cross-cutting properties (`Commutative`, `Associative`)
- Serialize expressions to Python syntax or RPN; test equivalence exhaustively

An AST (abstract syntax tree) is a data structure which represents an expression of some sort. The expression might be arithmetic, logical, or even something as complex as an entire computer program. We need to be able to programmatically manipulate these data structures in order to evaluate, simplify, compile, and analyze expressions.

An AST is often specified by an ADT (algebraic data type) which completely describes what the shape of the AST is allowed to be. An ADT allows us to detect syntax errors or other malformations in an AST,

as well as provides a systematic way to manipulating an AST.

5.1 N-Ary Trees

ASTs and ADTs are both implemented as n-ary trees. An n-ary tree is like a binary tree (Chapter 3) except that an internal node (parent) may have 1 or more child nodes. Sometimes certain `classes` of nodes necessarily have a fixed number of children; in other cases nodes are allowed to have a variable number of children depending on the context. The *syntax* of an AST is sometimes specified by a corresponding ADT (algebraic data type); other times the ad-hoc *syntax* is simply enforced by the program itself in terms of run-time error checking.

5.2 Arithmetic Expressions

We will look at two different approaches for representing arithmetic expressions. In Section 5.2.1 we will look at using the built in `list` and `tuple` data types to, and in Section 5.3.1 we will see how OO makes the simple tasks somewhat more difficult, but it allows the problem to be generalized in a way which makes many more complex problems easier.

5.2.1 Expressions with Fundamental Data Structures

We start by representing an expression, $(3 - \frac{20}{5}) \times (4 + 7)$, whose AST, Abstract Syntax Tree, is shown in Figure 5.1 as a tree of Python lists and tuples.

```
1 exp = ('*', ('-', 3,
2                   ('//', 20, 5)),
3             ('+', 4, 7))
```

These types of ad-hoc data structures are often called *s-expressions* or *sexps* in languages such as lisps. However in the Python language the term s-expression is rarely used. We have to use strings to indicate operators as there's really no other choice in the Python language. We define a variable named 'evaluators' which makes explicit all the operations we

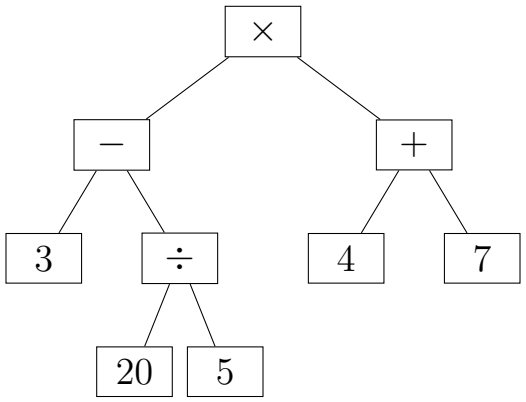


Figure 5.1: Abstract syntax tree representing expression $(3-\frac{20}{5})\times(4+7)$

support along with an anonymous function for each which specifies its semantics.

The use of these ad-hoc data structures (built-in lists, tuples, dictionaries) has advantages and disadvantages. One advantage is that no declarations are necessary. You are free to use expressions such as `exp = ('*', ...)` above as the Python reader already fully support it, and you may pass this kind of object around within your program and parse them using built-in operations such as array slicing.

A disadvantages of these data structures is that Python does not understand their semantics. It is difficult to determine whether the user has followed *the rules* for creating such objects. For example `exp = ('//', ('+', 1, "hello"), 12)`, does this expression make sense? Python believes it is a perfectly valid tuple, but it probably does not make sense for the application you are implementing.

5.2.2 Evaluation

```
evaluators = {'+': (lambda x, y: x + y),
              '-': (lambda x, y: x - y),
              '*': (lambda x, y: x * y),
              '/': (lambda x, y: x / y),
              '%': (lambda x, y: x % y),
              '//': (lambda x, y: x // y)
              }
```

Now we define a function `eval_expr` which takes an expression and returns the number it *evaluates* to.

```python
def eval_expr(exp):
  if not isinstance(exp, tuple):
    return exp
  op, a, b = exp
  if op not in evaluators:
    raise ValueError(f"unknown operator: {op}")
  else:
    return evaluators[op](eval_expr(a), eval_expr(b))
```

It is fairly easy to extend this code to support other binary operators. For example, to support the power operator x^y we could simply add the line `'**': (lambda x, y: x ** y),` to `evaluators`. However, it is not clear how to add unary operators such as `'-'` for two reasons: (1) this character is already *registered* as a binary operator, and (2) the code doesn't support unary operators—it assumes all operators are binary.

5.2.3 Equivalence

How can we know whether two expressions are equivalent? For example: for numerical arithmetic we know that $a + (b + c) = (a + b) + c$, and for Boolean algebra we know $a \wedge (b \vee c) = (a \vee b) \wedge (a \vee c)$. But in general it is a difficult problem to decide the equivalence of two expressions.

If the set of values for each variable is finite, we can arduously test the two given expressions all inputs and assert that the outputs are equal. We can do this for modulo arithmetic, and for Boolean values.

Write code to represent algebraic expressions in a finite number of variables. Assume we are using modulo arithmetic. Use exhaustive search to decide whether two given expressions are equivalent.

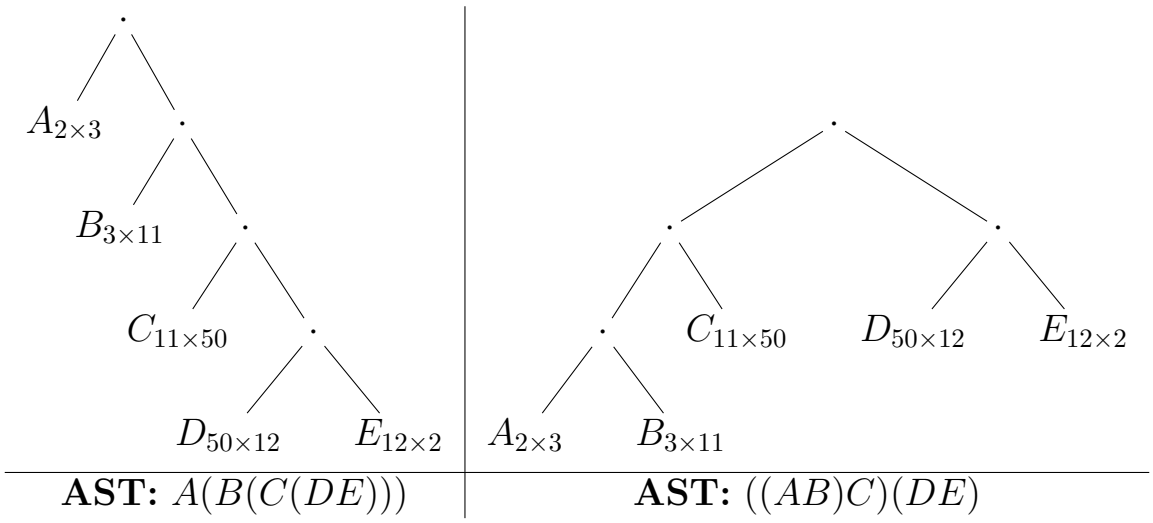
Another approach for detecting equivalence is to attempt to convert the two expressions to the same `canonical` form; *i.e.*, simplify them to the same syntactic structure.

5.2.4 Associativity of Matrix Multiplication

The problem we investigate here is given a chain of matrices of different but compatible sizes, how can we minimize the number of floating point multiplications (flops) necessary to compute the complete matrix multiplication. For example given matrices of the following sizes: $A \in \mathbb{R}^{2\times3}$,

$B \in \mathbb{R}^{3\times 11}$, $C \in \mathbb{R}^{11\times 5}$, $D \in \mathbb{R}^{5\times 12}$, and $E \in \mathbb{R}^{12\times 2}$, which of the following is more efficient to compute: $A(B(C(DE)))$ or $((AB)C)(DE)$?

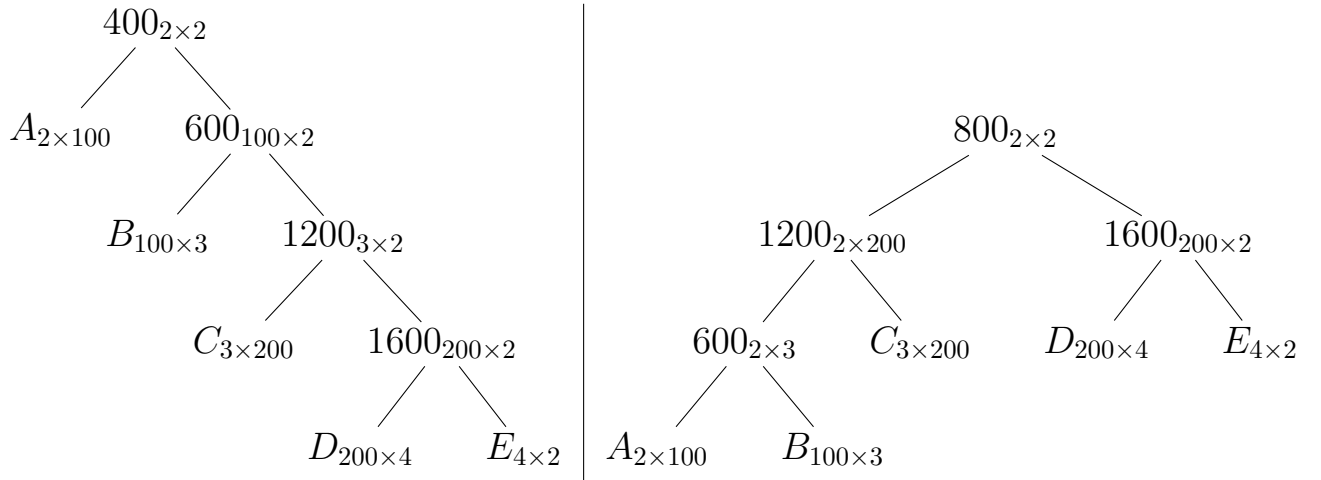
These two expressions represent the exact same mathematical value and the same final theoretical result of a 2×2 matrix. However, they are represented by different ASTs.



AST: $A(B(C(DE)))$ **AST:** $((AB)C)(DE)$

Let's calculate how many floating point operations are necessary to compute each of these matrix products. Recall that to multiply two matrices of size $n \times k$ and $k \times m$ to obtain a matrix of size $n \times m$ we need to compute $n \times m$-many entries, and each entry requires k-many multiplications and $k-1$-many additions. We ignore the additions. Thus computing the $n \times m$ matrix requires $n \times k \times m$-many multiplications. In the following figure leaf nodes are labeled A, B, ..., C with their respective sizes. Internal nodes are labeled such as $396_{3\times 2}$, meaning that a matrix of size 3×2 was obtained by multiplying two matrices of sizes $3 \times k$ and $k \times 2$, and $3 \times k \times 2 = 396$ multiplications were performed to obtain this 3×2 matrix. Thus to count the total number of flops needed on each side, we simply add up the numbers labeling the internal nodes, ignoring their subscripts.

Computing $A(B(C(DE)))$ requires $400+600+1200+1600 = 3800$ total flops, while computing the same matrix by multiplying $((AB)C)(DE)$ requires $800 + 1200 + 1600 + 600 = 4200$ flops. This is a difference of $\frac{4200-3800}{3800} = 10.5\%$.



AST: $A(B(C(DE)))$
Multiplications: 3800

$((AB)C)(DE)$
4200

This matrix multiplication can be done using only 3036 floating point multiplications— *i.e.*, 1064 fewer flops than the worst of the two shown above. The student is challenged to discover how to draw the parentheses to obtain this reduction.

For a given chain of matrices, how can we choose the AST which results in the fastest matrix multiplication? This problem is known as the *matrix chaining problem*.

Notice that the two ASTs list the same matrices as leaf nodes, and in the same left-to-right order. Thus to solve the matrix chaining problem for a given list of matrices of given sizes, we *simply* need to find which matrices to make the left child of the top node, and which to make the right child, and then solve the same problem recursively on the left and right child nodes. *I.e.*, we need to find the pivot point at the top of the tree, and recursively thereunder.

A classical approach to solving this problem is with a technique called *dynamic programming*. We will not discuss dynamic programming here. Instead, first we will present a way to perform this tree construction computation in a horribly inefficient way. Then, we'll look at how to make a small change to make the computation time-efficient.

Here is the idea. We wish to compute $ABCDE$. The optimum process will include a sequence of multiplications ending with one of the following, and we would like to know which of these four alternatives is the best.

1. $A_{2\times100} \cdot (BCDE)_{100\times2}$
2. $(AB)_{2\times3} \cdot (CDE)_{3\times2}$
3. $(ABC)_{2\times200} \cdot (DE)_{200\times2}$
4. $(ABCD)_{2\times4} \cdot E_{4\times2}$

However, we cannot directly compute which of these four alternatives is the best, because we need to know the most efficient way to compute the following:

1. $(BCDE)_{100\times2}$
2. $(AB)_{2\times3}$ — 600 flops
3. $(CDE)_{3\times2}$
4. $(ABC)_{2\times200}$
5. $(DE)_{200\times2}$ — 1600 flops
6. $(ABCD)_{2\times4}$

There is only one way to compute $(AB)_{2\times3}$, which is $2 \times 100 \times 3$ flops; and for $(DE)_{200\times2}$, it is $200 \times 4 \times 2$ flops. But we can compute the other three using the same kind of recursive search.

Suppose at this point we have computed these sub-problems and we have:

1. α_{BCDE} = optimum number of flops of $(BCDE)_{100\times2}$
2. α_{AB} = optimum number of flops of $(AB)_{2\times3}$ — 600 flops
3. α_{CDE} = optimum number of flops of $(CDE)_{3\times2}$
4. α_{ABC} = optimum number of flops of $(ABC)_{2\times200}$
5. α_{DE} = optimum number of flops of $(DE)_{200\times2}$ — 1600 flops
6. α_{ABCD} = optimum number of flops of $(ABCD)_{2\times4}$

Then knowing $\alpha_{BCDE} \ldots \alpha_{ABCD}$, we can compute the optimum number of flops to compute $ABCDE$ by computing the minimum of the following:

$$\begin{aligned}
A_{2\times100} \cdot (BCDE)_{100\times2} &\implies \alpha_{BCDE} + (2 \times 100 \times 2) \\
(AB)_{2\times3} \cdot (CDE)_{3\times2} &\implies \alpha_{AB} + \alpha_{CDE} + (2 \times 3 \times 2) \\
(ABC)_{2\times200} \cdot (DE)_{200\times2} &\implies \alpha_{ABC} + \alpha_{DE} + (2 \times 200 \times 2) \\
(ABCD)_{2\times4} \cdot E_{4\times2} &\implies \alpha_{ABCD} + (2 \times 4 \times 2)
\end{aligned}$$

The following Python function, `best_chain`, computes the minimum (or maximum) number of flops needed to multiply a chain of matricies, given a list of 2-tuples specifying the dimensions of the matricies in question. The local function, `best`, computes the value of α in the above example.

```
from functools import cache

def best_chain(opt, dims:List[Tuple[int,int], ...]) -> int:

    @cache
    def best(dims) -> int:

        if len(dims) == 1:
            return 0
        elif len(dims) == 2:
            (n,k),(k2,m) = dims
            assert k == k2
            return n * k * m
        else:
            m,_ = dims[0]
            _,n = dims[-1]
            return opt(best(left) + (m * k * n) + best(right)

                       for pivot in range(1, len(dims))
                       for left in [dims[0:pivot]]
                       for right in [dims[pivot:]]
                       for k,_ in [right[0]]
                       )

    return best(tuple(dims))
```

The function `best_chain` can be called as follows:

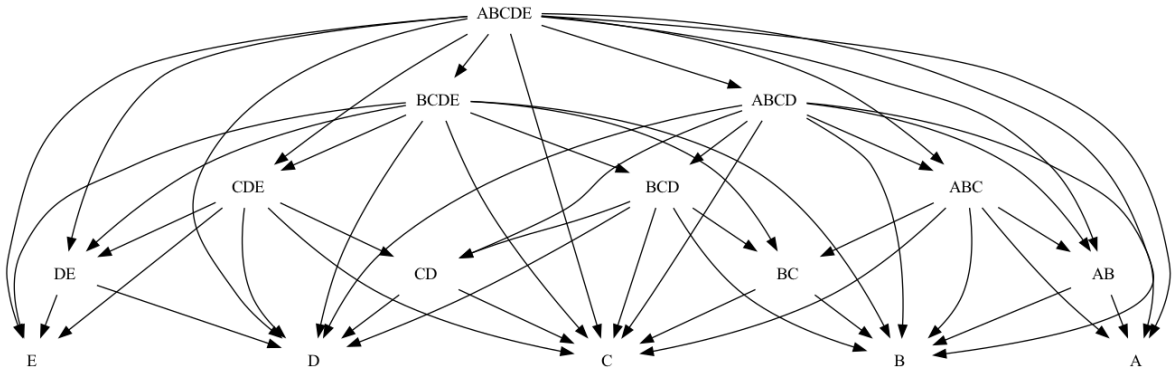


Figure 5.2: Recursive calls for computing chaining of ABCDE

```
best_chain(min, [(2, 100), (100, 3), (3, 200), (200, 4), (4, 100)])
 ==> 3824

best_chain(max, [(2, 100), (100, 3), (3, 200), (200, 4), (4, 100)])
 ==> 2160000
```

The local `best` calls itself many times, and often with the same arguments. For example to compute the chaining for $ABCDE$, it must compute (among others) the best chaining for AB. Also when it computes the best chaining for $ABCD$ it must compute the best chaining for AB. Again when it computes the best chaining for ABC it must yet again compute the best chaining for AB. The graph in Figure 5.2 shows how the same results are computed redundantly. However, of all the calls to `best`, There are only 15 with different arguments as shown in Figure 5.2. If we run the `best` function is called 45 times, but with the caching it is only called 15 times, once for each distinguishable input.

The annotation `@cache` is used because the local function would otherwise be called exceedingly many times with the exact same argument list. This `@cache` annotation allows the function to cache its input/output values in a hidden dictionary and whenever the function is called a second time, a simple dictionary lookup is done. Because the function input values must be valid hash keys, we call `best` on the bottom line of `best_chain` with `best(tuple(dims))` to assure that the input value of `best` is always a tuple, and thus a valid dictionary key.

If we call the function on a list of dimensions longer than 5, so with 10 we see that the function is called 10935 times, but with caching it is

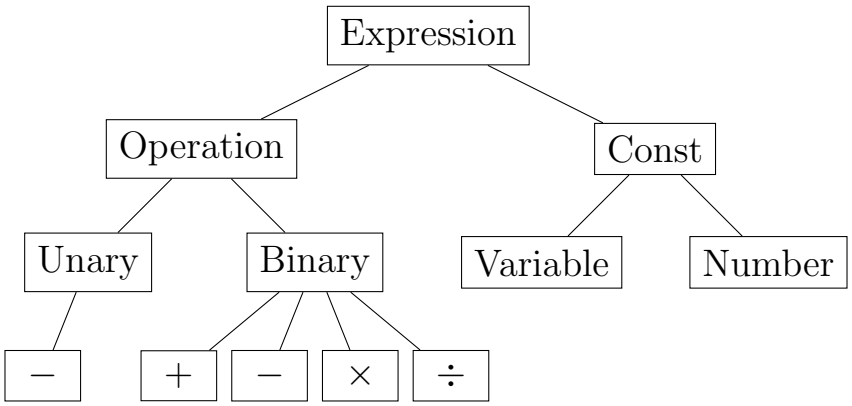


Figure 5.3: ADT, Algebraic Data Type

called only 55 times.

5.3 Algebraic Data Type

An ADT (algebraic data type) is a tree which exhaustively describes or specifies the syntax for an AST. An example is shown in Figure 5.3.

Sometimes it may not be clear which form of ADT to use. For example, we could group the binary operations in Figure 5.3 into operations which are commutative ($+$ and $\times$) and those which are not commutative ($-$ and $\div$). We must consider whether making the ADT more specific simplifies the resulting code or makes it more complex.

5.3.1 Expressions with Object Orientation

To represent an expression in an object oriented way there are two trees we need to consider: the tree representing an expressions, and a tree representing the semantics. A node in the AST represents an operation such as addition or multiplication; a node in the semantic tree represents the union of a set of behavior. The semantic tree is implemented in Python as a hierarchy of classes, each of which represents an is-a relationship. For example, addition is a binary operation, multiplication is also a binary operation, but negation is unary operation, binary and unary operations are both arithmetic operations. A number and a variable are both values, but they are not operations. We need a hierarchy of classes

which represents these relations as super-class sub-class relations.

5.3.2 ADT as Class Definitions

The ADT in Figure 5.3 can be implemented in Python using class definitions to express the is-a relations between the types. First the abstract types.

```
class Expression:
  pass

class ExpOperation (Expression):
  pass

class ExpConst (Expression):
  pass

class ExpUnary (ExpOperation):
  pass

class ExpBinary (ExpOperation):
  pass
```

Now we define the leaf level classes. These are the classes which users are intended to instantiate.

```
class ExpVariable (ExpConst):
  pass

class ExpNumber (ExpConst):
  pass

class ExpMinus (ExpUnary):
  pass

class ExpAdd (ExpBinary):
  pass

class ExpSubtract (ExpBinary):
  pass

class ExpMultiply (ExpBinary):
  pass

class ExpDivide (ExpBinary):
  pass
```

Several methods need to be implemented. We should attempt to im-

plement the methods on least specific classes as possible. If we find ourselves duplicating code in different methods, we must ask ourself whether the class hierarchy is correct.

We look again at how to represent a mathematical expression such as $(3 - \frac{20}{5}) \times (4 + 7)$, but this time using classes.

```
exp = ExpMultiply(ExpSubtract(ExpNumber(3),
                              ExpDivide(ExpNumber(20),
                                        ExpNumber(5))),
                  ExpAdd(ExpNumber(4), ExpNumber(7)))
```

This is more complex syntax than s-expression syntax discussed in Section 5.2.1. However, the constructors for each class assure that the syntax is correct, otherwise the construction fails.

5.3.3 Multiple Inheritance

One tool for helping to limit duplicate code in methods is multiple inheritance. For example some of the binary operators are commutative, and some are associative. Python allows us to declare abstract classes (whose syntax is exactly the same as ordinary classes), and mix them in to subclasses.

For example we might define the classes `Commutative` and `Associative` as follows.

```
class Commutative (ExpBinary):
  pass

class Associative (ExpBinary):
  pass
```

Having these classes, we could refactor the operators as follows:

```
class ExpAdd (ExpBinary, Commutative, Associative):
  pass

class ExpSubtract (ExpBinary):
  pass

class ExpMultiply (ExpBinary, Commutative, Associative):
  pass

class ExpDivide (ExpBinary):
  pass
```

5.3.4 Added a new Operator

Imagine we have already defined an ADT implementing several arithmetic operations, but we would like to add a new operator. For example, suppose we would like to add the exponent operator so we can compute x^y.

If the ADT is implemented as in Section 5.3.2, then adding the new operator is straight forward. We create a new subclass of `ExpBinary` as follows, and fill in the details by implementing some or all the methods specializing on this new class. Note that exponentiation is neither a commutative nor associative operation.

```
class ExpExponent (ExpBinary):
  pass
```

5.4 Representing Expressions containing Variables

An expression might contain variables. The presence of variables makes some operations more complex. For example, sub-trees having no variable are still subject to constant folding, but other subtrees are not. Nevertheless, other simplifications such as distributive laws may still be possible.

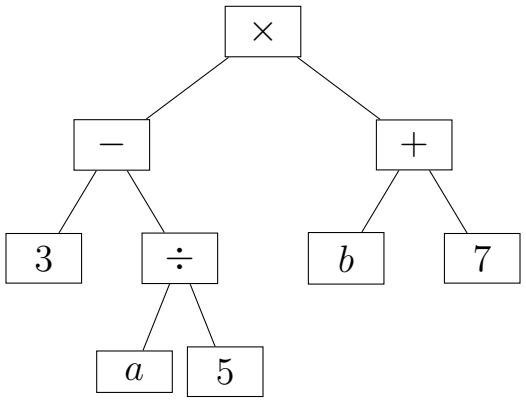


Figure 5.4: AST of expression with variables $(3 - \frac{a}{5}) \times (b + 7)$

$$a \times (b + c) \rightarrow (a \times b) + (a \times c)$$
$$(a \times b) + (a \times c) \rightarrow a \times (b + c)$$

Some simplifications are possible in the presence of variables, even without access to an environment. For example, $a - a$ simplifies to zero, even if the value of a is unknown.

To evaluate an expression containing variables, an environment is necessary. An *environment* is a mapping of variable name to value. In Python an environment may be implemented as a dictionary.

Here is an example evaluation function for such a case. We assume there is a predicate `is_variable` which determines whether an object is a variable.

```
def eval_expr(exp, env):
  if is_variable(exp):
    return env[exp]
  if exp not in env:
    raise ValueError(f"unbound variable: {exp}")
  if not isinstance(exp, tuple):
    return exp
  op, a, b = exp
  if op not in evaluators:
    raise ValueError(f"unknown operator: {op}")
  else:
    return evaluators[op](eval_expr(a, env),
                          eval_expr(b, env))
```

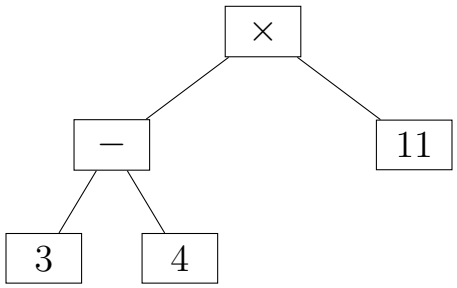


Figure 5.5: Abstract syntax tree representing expression $(3-4)\times 11$

5.5 AST manipulation

In some cases we may be able to transform an AST into another tree representing an equivalent expression. for example, the AST in Figure 5.5 represents the expression $(3-4)\times 11$ which can be considered equivalent to $(3-\frac{20}{5})\times(4+7)$ from Figure 5.1.

This type of simplification is called constant propagation. If the child nodes of an operator are all numbers, then we can replace the node with an equivalent number, *e.g.* replacing $\frac{20}{5}$ with 4.

Once the AST has been *simplified* as in the transition from Figure 5.1 to Figure 5.5, further simplifications may become evident. $(3-4)\times 11$ can be simplified to -1×11, and -1×11 may be simplified to -11. Finally -11 cannot be further simplified, or otherwise stated -11 simplifies to itself. *I.e.*, -11 is the fixed point of the simplification procedure.

To completely simplify an AST, we must continue simplifying the resulting AST until a fixed point is reached. This typically means there is a set of simplification rules.

Creating a set of *correct* simplification rules can be tricky. How do we know there is not a loop in the rules. For example if there are two rules as follows, then there is a loop in the simplification and a fixed point might never be reached.

$$a\times(b+c)\rightarrow(a\times b)+(a\times c)$$
$$(a\times b)+(a\times c)\rightarrow a\times(b+c)$$

5.6 Boolean Expressions

There is a homework assignment, `ast_bool.py` to implement Boolean arithmetic using this same programming model. In that assignment you'll need to implement the logical complement operator as a unary operator. Rather than allowing integers, the only supported constants are `True` and `False`.

Write code to represent Boolean expressions in a finite number of variables. Use exhaustive search to decide whether two given expressions are equivalent.

5.7 Serializing an Expression Tree

5.7.1 Serializing as Python syntax

We first implement the function `to_python` assuming an expression uses the s-expression convention presented in Section 5.2.1. This function prints a string which we can copy and paste back into the Python interpreter to check whether the `eval_expr` works.

```
def to_python(exp):
  if not isinstance(exp, tuple):
    return str(exp)
  op, a, b = exp

  if op not in evaluators:
    raise ValueError(f"unknown operator: {op}")
  return "(" + to_python(a) + op + to_python(b) + ")"
```

If we test the code, `eval_expr(exp)` returns `-11` and `to_python(exp)` returns the string, `'((3-(20//5))*(4+7))'`. If we copy past that output back into the Python console we see that it also evaluates to `-11`.

As a second approach we show an excerpt of the implementation of `to_python` assuming an Object Orientation implementation as shown in Section 5.3.2. A more complete implementation can be find in the homework exercise `ast_arith.py`.

```
class ExpNumber (ExpConst):

    def to_python(self):
        return f"{self.value}"


class ExpVariable (ExpConst):

    def to_python(self):
        return self.name


class ExpBinary (ExpOperation):

    def to_python(self) -> str:
        return "(" + self.left.to_python() \
                   + ' ' + self.op + ' ' \
                   + self.right.to_python() + ")"

class ExpUnary (ExpOperation):

    def to_python(self) -> str:
        return "(" + self.op + self.expr.to_python() + ")"
```

5.7.2 Serializing as RPN

We can take an expression tree as input and output a list of tokens compatible with the `rpn.rpn` function.

The code for `to_rpn` looks very similar to `to_python` except that the operator: `+`, `-`, etc is printed after the arguments, and no parentheses are necessary. See the homework exercise `ast_arith.py` for details.

5.8 Homework

There are two homework exercises for this chapter. The first one `ast_bool.py` whose tests are in `ast_bool_test.py`, is to implement an AST to represent Boolean expressions with variables. Boolean operations may use operators AND, OR, NOT, EQUAL, and IMPLIES.

The second homework problem is `ast_arith.py`. This problem is different than the others in the course. In this problem you are given code with a bug, and with insufficient tests in `ast_arith_test.py`. You task is to find the bug and write good test cases to identify the bug.

Chapter 6

Finite Automata

Chapter Objectives

- Represent a finite automaton as (Σ, Q, I, F, T); distinguish NFA from DFA
- Match strings with a DFA in $O(|w|)$; with an NFA by tracking sets of states
- Store transitions as dict-of-dicts for $O(1)$ next-state lookup
- Complement a DFA: complete it first, then flip accepting states
- Apply the fold/`reduce` pattern to express automaton traversal concisely
- Enumerate a regular language by partitioning into finite length-k subsets

6.1 Wild Cards

In the UNIX shell it is possible to specify a set of file names using so-called *wild cards*. *E.g.*, `ls *.py` will list all files in the current directory ending with the suffix `.py`. The shell itself recognizes that wild cards are being used and before it executes the `ls` command, it first *expands* the wild card and executes the `ls` command with a sequence of file names. It is important to note that when `ls` is run, it does not know that a wild

card was used—it simply sees a sequence of file names. So the commands `echo *.py` and `rm *.py` also behave the same way.

6.2 Regular Expressions

Regular expressions are similar to wild cards but the syntax is more elaborate and they don't *expand* to or `match` only file names, but rather each regular expression `matches` a possibly infinite set of strings. This set of matching strings is called the `language` of the regular expression.

Regular expressions have a mathematical form (Section 6.2.3) and a serialized form used in programming languages. The particular programmer syntax depends on the programming language and the regular expression library being used. Some programming languages have several different regular expression libraries which can be used, and some programming languages have built-in support. Two common but contradictory standards are POSIX regex and PCRE regex (Perl-compatible regular expressions).

ATTENTION: the syntax used by one regular-expression library may or may not work with another library.

6.2.1 The `grep` command

The UNIX `grep` command will print all the lines of a named file which match a given regular expression.

```
> grep for ads.py
# Custom exception for test timeout
      for path in os.environ["PATH"].split(os.pathsep):
  comment_lines = [line for line in lines
                   for sline in [line.lstrip()]
  blank_lines = [line for line in lines
      return [copy_data(x) for x in a]
      for k in range(len(ar1)):
      for pyfile in pyfiles:
      for pyfile in pyfiles:
      for pyfile in pyfiles:
                    for line in lines
      for (file, lineno, chno, culprit, pep, line) in parsed:
      """Check␣for␣certain␣illegal␣imports."""
      for statement in fp.readlines():
      for tabu in ["math.factorial",
```

We can limit (reduce) the output by using a more restrictive regular expression

```
> grep 'for.*p.*in' ads.py
   for path in os.environ["PATH"].split(os.pathsep):
   for pyfile in pyfiles:
   for pyfile in pyfiles:
   for pyfile in pyfiles:
     for (file, lineno, chno, culprit, pep, line) in parsed:
   for statement in fp.readlines():
```

Or limit even further with a more restrictive regular expression.

```
> grep 'for.*py.*in' ads.py
   for pyfile in pyfiles:
   for pyfile in pyfiles:
   for pyfile in pyfiles:
```

6.2.2 The Python `re` library

The documentation for Python regular expressions can be found here: `https://docs.python.org/3/library/re.html`.

The Python `re.match` function is used to determine whether a given string matches a given regular expression. However, Python's concept of matching means *matches the beginning of the string*. This is different than the semantics of `grep`-style regular expressions.

```
1  import re
2
3  >>> re.match("a", "bbbabbb")
4  None
5
6  >>> re.match(".*a.*", "bbbabbb")
7  <re.Match object; span=(0, 7), match='bbbabbb'>
8
9  >>> re.match(".*a", "bbbabbb")
10 <re.Match object; span=(0, 4), match='bbba'>
11
12 >>> re.match("^.*a$", "bbbabbb")
13 None
14
15 >>>re.match("^.*a.*$", "bbbabbb")
16 <re.Match object; span=(0, 7), match='bbbabbb'>
```

The expression `re.match("a", "bbbabbb")` tests whether `"bbbabbb"` *BEGINS* with the string `"a"`.

The expression `re.match(".*a.*", "bbbabbb")` tests whether `"bbbabbb"` *CONTAINS* the string `"a"`.

The expression `re.match(".*a", "bbbabbb")` tests whether some prefix of `"bbbabbb"` *CONTAINS* the string `"a"`.

The expression `re.match(".*a$", "bbbabbb")` tests whether `"bbbabbb"` *ENDS* the string `"a"`.

Let's create a Python regular expression which will determine whether a given string represents an integer. It should accept a string like `"2543"` and should reject a string like `"2543.23"` or `"xyz"`.

There are lots of corner cases to consider. In each case, you the programmer, must decide whether the string should be accepted or rejected as an integer-string: `"0"`, `"00"`, `"0123"`, `" 123"`, `"-123"`, `"--123"`, `"++123"`, `"+-123"`, `"+0"`, `"-0"`.

Here is a suggested solution. Other solutions are possible.

```
>>> r = re.compile("^[-]?(0|([1-9][0-9]*))$")
>>> re.match(r,"-01")
None

>>> re.match(r,"--0")
None

>>> re.match(r,"123")
<re.Match object; span=(0, 3), match='123'>

>>> re.match(r,"-123")
<re.Match object; span=(0, 4), match='-123'>

>>> re.match(r,"-123.3")
None

>>> re.match(r,"-0123")
None

>>> re.match(r,"-123")
None
```

6.2.3 Regular Expressions Formally

Given an alphabet such as $\Sigma = \{a, b, c\}$, the set of all words of finite length consisting of only letters from Σ is referred to as Σ^*. The set Σ^* has infinite cardinality, but each word $w \in \Sigma^*$ has finite length. For

example

$$\begin{aligned} a &\in \Sigma^* \\ aa &\in \Sigma^* \\ aba &\in \Sigma^* \\ abbbcab &\in \Sigma^* \\ aaabbbaaccccabac &\in \Sigma^* \\ &\vdots \end{aligned}$$

There is a special word called the *empty word* and designated ε. Note that $\varepsilon \in \Sigma^*$ because it has finite length and only consists of letters from Σ, precisely zero such letters. So $\varepsilon \in \Sigma^*$ but $\varepsilon \notin \Sigma$.

We may define the set of all regular expressions by defining a set of symbols and designated the set of words (a subset of Σ^*) which each such symbol corresponds to.

Trivial Regular Expression	**Set of Words**
ε	The set containing only the empty word.
Σ	The set of single letter words, with letters coming from Σ
ℓ	If $\ell \in \Sigma$, then this regular expression corresponds to the singleton set of the one letter word whose letter is ℓ.

The *language* of a regular expression is the set of words the regular expression matches. Mathematically we denote the language of regular expression r as $[\![r]\!]$.

We also define the following set of recursive regular expressions, assuming f and g are regular expressions whose corresponds languages are $[\![f]\!]$ and $[\![g]\!]$, then the following are regular expressions.

Given two languages, A, B, we define the concatenated language $A \cdot B$ (sometimes abbreviated AB) as the set of all concatenations xy where

$x \in A$ and $y \in B$. *E.g.*,

$$\begin{aligned} A &= \{a, ab\} \\ B &= \{b, bb, bc\} \\ A \cdot B &= \{ab, abb, abc, abb, abbb, abbc\} \\ B \cdot A &= \{ba, bab, bba, bbab, bca, bcab\} \\ A \cdot A &= \{aa, aab, aba, abab\} \end{aligned}$$

Notice that $(AA)A = A(AA)$:

$$\begin{aligned} (A \cdot A) \cdot A &= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\} \\ A \cdot (A \cdot A) &= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\} \end{aligned}$$

We can also use exponent notation with languages.

$$\begin{aligned} A^0 &= \{\varepsilon\} \\ A^1 &= A \\ A^2 &= A \cdot A \\ A^3 &= A^2 \cdot A \\ \vdots &= \vdots \\ A^n &= A^{n-1} \cdot A \end{aligned}$$

We can also define the *Kleene star*.

$$\begin{aligned} A^* &= \bigcup_{n=0}^{\infty} A^n \\ &= \{w \in \Sigma^* \mid \exists n \in \mathbb{N}, w \in A^n\} \end{aligned}$$

Set concatenation allows us to define concatenated regular expressions, and other commonly used regular expression operators: $*$, $+$, &, !, and ?.

Compositional Regular Expression	Language of the Regular Expression
$f \cdot g$	$[\![f]\!] \cdot [\![g]\!]$
fg	same as $f \cdot g$
$f + g$	$[\![f]\!] \cup [\![g]\!]$
$f \,\&\, g$	$[\![f]\!] \cap [\![g]\!]$
$!f$	$\Sigma^* \setminus [\![f]\!]$
f^*	$[\![f]\!]^*$
f^+	$f \cdot f^*$
$f^?$	$f + \varepsilon$

Regular expressions are often written using parentheses for grouping such as: $((ab^*) + (a^*b))^+$

6.2.4 Regular Expression Language Enumeration

A set can be defined in one of two way. (1) by enumerating it, (2) by a predicate. By predicate we mean define an algorithm (function, procedure) which when given a perspective element of the set, the algorithm should decide (true or false) whether the given element is in the set. We discuss such a predicate function starting in Section 6.8. For now we quickly discuss the problem of enumeration.

The language of a regular expression is a enumerable set. This means given a regular expression r, we can define a bijection from $\mathbb{N} \to [\![r]\!]$. Constructing the bijections can be tricky.

Example: if $r = a^*$, then we can easily enumerate the language $[\![a^*]\!]$ as: $0 \to \varepsilon$, $1 \to a$, $2 \to aa$, $3 \to aaa$, ...

How can we enumerate $r = a^*b^*$? If we try the same ordering as before: $0 \to \varepsilon$, $1 \to a$, $2 \to aa$, $3 \to aaa$, ..., then we never arrive at b, bb, ab, etc.

What we have to do is recognize that we can partition $[\![a^*b^*]\!]$ (or in fact the language of any regular expression) into an enumerated set of finite subsets: first the subset of words of length 0, next the subset of words of length 1, then the subset of words of length 2, etc.

$$[\![a^*b^*]\!] = \bigcup_{k=0}^{\infty} P_k \,,$$

where $P_k = \{w \in [\![a^*b^*]\!] \mid |w| = k\}$. Since P_k has finitely many elements, it can be alphabetized into dictionary order.

Using this so-called *military order* we can enumerate $[\![a^*b^*]\!]$ as:

- $0 \to \varepsilon$
- $1 \to a$, $2 \to b$
- $3 \to aa$, $4 \to ab$, $5 \to bb$.
- $6 \to aaa$, $7 \to aab$, $8 \to abb$, $9 \to bbb$.
- $10 \to aaaa$, $11 \to aaab$, $12 \to aabb$, $13 \to abbb$, $14 \to bbbb$.
- ...

6.2.5 Equivalence

Deciding whether two regular expressions are equivalent is a difficult task. In general there are possibly infinitely many regular expressions with the same language. *E.g.*, a very simple example: aa^* and a^*a match exactly the same strings. Another more elaborate example do $(a^?b + ab^?)^*$ and $(a^*b^*)^*$ have the same language? What about $(a^*ba)^*$ vs $(a^*b)^*$?

One of the programming problems related to regular expressions is knowing how to refactor them. If you are given a program which someone else wrote, and you have to fix a bug or add a new feature, you may want to rewrite or extend a regular expression. It is very difficult through simple reasoning whether the changes you have made have the effect you desire.

It is easy to imagine that as we looking at more and more complex regular expressions, it is more and more difficult to determine whether they are the same. In general given two expressions r_1 and r_2 whose respective languages are L_1 and L_2 any of the following might be possible.

- $L_1 \subset L_2$
- $L_2 \subset L_1$
- $L_2 = L_1$
- $L_2 \cap L_1 = \emptyset$
- $\overline{L_2 \cup L_1} = \emptyset$

Sometimes it is possible to determine which of these relations hold using algebraic manipulation and logic. However, the most common way to answer these questions is using Finite Automata as we will see in the upcoming sections.

6.2.6 How a regular expression works

Matching a string against a regular expression has two phases.

1. Convert the regular expression to a finite automaton (in Python, `re.compile(...)`).
2. Walk the finite automaton by iterating over the characters in the string, (in Python, `re.match(...)`).

A finite automaton has states, indicated as circles with a name inside; the name usually is a number. The finite automaton in Figure 6.1 has states 0, 1, 2, and 3. Some states, such as 0 and 1 in Figure 6.1, are so-called *initial* states. Initial states are denoted by an incoming arrow having no source state. These state Some states, such as 2 and 3 in Figure 6.1, are so-called *accepting* states, sometimes referred to as *final* states. Accepting states are denoted by a double circle. It is possible that a state be both an initial and simultaneously a final state, and it is possible that a state be neither initial nor final.

A *computation* is a sequence of transitions starting at an initial state. We label the arrows with the label identifying the transition, such as: $0 \xrightarrow{-} 1 \xrightarrow{3} 3 \xrightarrow{1} 3 \xrightarrow{0} 3 \xrightarrow{8} 3$ which is a path in-turn tracing the labels `-3108`. If the right-most state reached by a computation is an accepting state, we say the computation is *successful*.

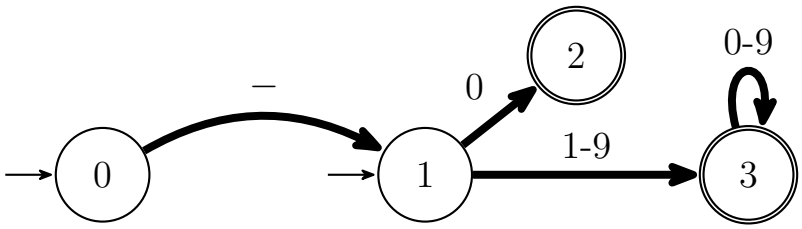


Figure 6.1: Non-deterministic Finite Automaton recognizing integer syntax. This finite automaton is non-deterministic because it has multiple initial states.

6.3 Non-deterministic Finite Automata

There are many finite automata which implement a given regular expression. The one shown here is non-deterministic. A non-deterministic may have more than one initial state, and there may exist a state which multiple transitions having the same label.

Once the finite automaton has been generated, we may ask whether a given string corresponds to a *successful computation* on the finite automaton. *I.e.*, does the string correspond to the labels of a path which connects an initial state to an accepting state. Looking at the finite automaton above we see that the string `"0"` corresponds to the path from state 0 to state 2 ($0 \to 2$). Additionally the string `"-3108"` corresponds to the path $0 \xrightarrow{-} 1 \xrightarrow{3} 3 \xrightarrow{1} 3 \xrightarrow{0} 3 \xrightarrow{8} 3$. On the contrary there is no path whose labels correspond to `"00123"`.

6.4 Determinism

Any non-deterministic finite automaton may be converted to a deterministic finite automaton (DFA) having the exact same language. The graph in Figure 6.2 [Top] is a determinized version of the NFA in Figure 6.1.

There are well-known algorithms for converting an NFA to a DFA, but we will not study these algorithms in this course.

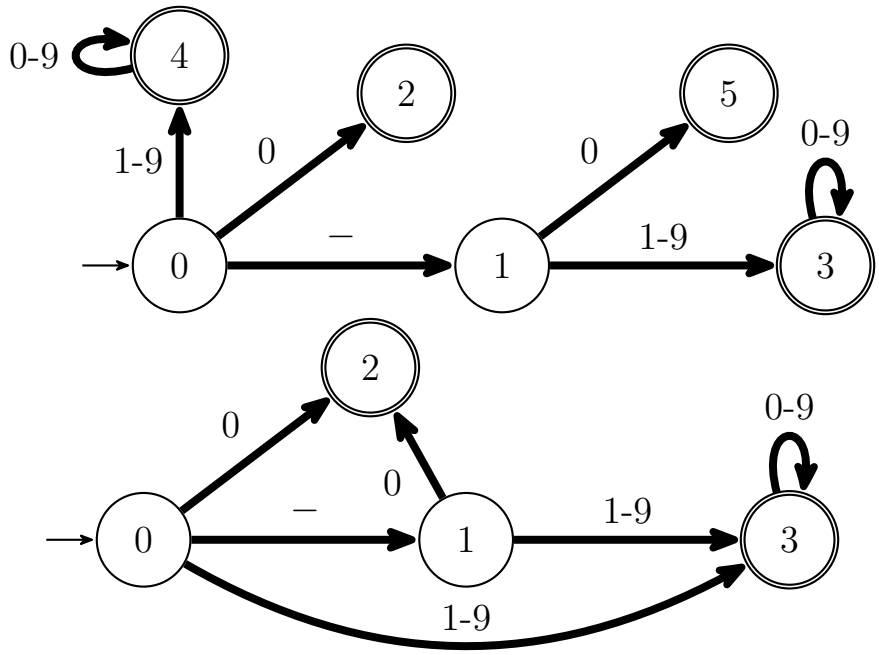


Figure 6.2: Deterministic Finite Automata recognizing integer syntax. [Top] non-minimized. [Bottom] minimized.

6.5 Minimization

Again, there are potentially many different DFAs which implement a given regular expression, but there is a unique DFA (modulo state-numbering) which is minimized.

The astute reader may notice that in the DFA in Figure 6.2 [Top], states 3 and 4 serve the exact same purpose, *i.e.* to match any number of digits 0 to 9. Furthermore, states 2 and 5 serve the same purpose, namely to accept the single digit 0, and to reject 0 followed by another digit. Since state 3 and 4 are *equivalent* they have been combined into the single state 3 in Figure 6.2 [Bottom]. Likewise, states 2 and 5 have been combined into state 2 in Figure 6.2 [Bottom].

The minimized DFA corresponding to the non-minimum DFA from Figure 6.2 [Top] can be seen in Figure 6.2 [Bottom].

There are well-known algorithms for minimizing a DFA, but we will not study these algorithms in this course.

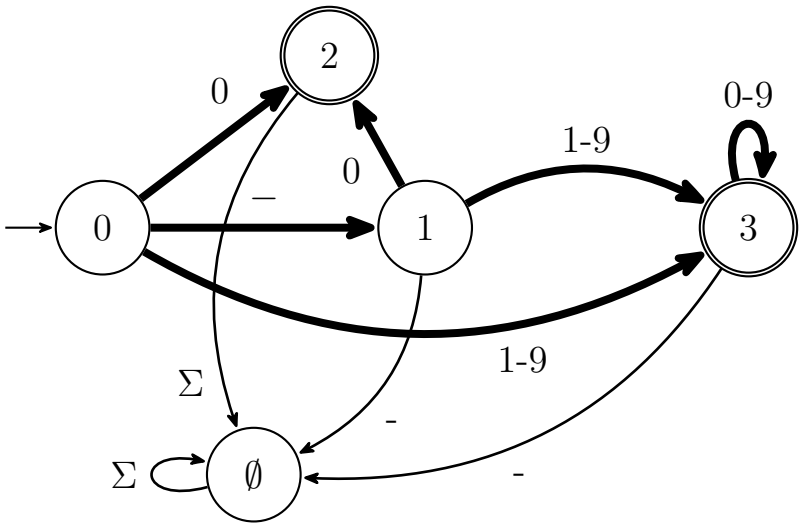


Figure 6.3: Complete DFA recognizing integer syntax. A sink state ∅ has been added.

6.6 Complete Finite Automata

You may notice that the DFA in Figure 6.2 [Bottom] (and other figures above) are incomplete in the sense that even though the alphabet (the set of valid characters) is `{"-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" }`, some states lack certain transitions. For example, state 0 has 11 transitions exiting it; however, state 1 lacks a transition labeled `"-"`, and state 2 has no exiting transitions at all. A finite automaton is called `complete` if every state has a transitions labeled by each letter of alphabet in question.

It is straightforward to extend any incomplete finite automaton (deterministic or otherwise) to be complete. To do this, we simply add a so-called *sink* state, and replace all missing transitions with a transition to this sink state. The graph in Figure 6.3 shows a complete version of Figure 6.2 [Bottom]. Notice that this graph is cluttered and more difficult to read. Because of this annoyance, the sink state is usually omitted in the drawing, even if it is present in the programmatic representation.

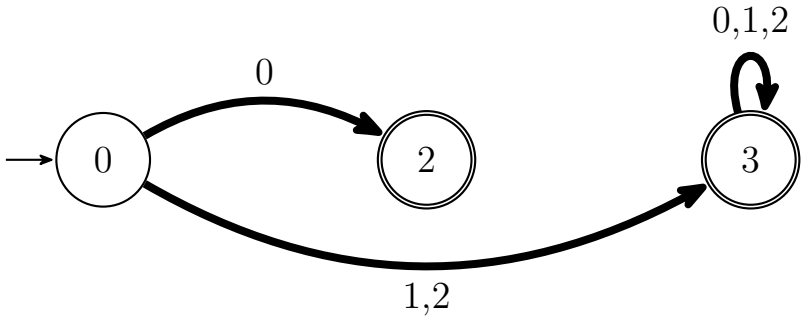


Figure 6.4: Subgraph of 6.2 [Bottom] recognizing positive integers in base-3.

6.7 Representing a Finite Automaton

A finite automaton (deterministic or otherwise) can be represented mathematically as a tuple $A = (\Sigma, Q, I, T)$ where Σ is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of accepting (final) states, and $T \subseteq Q \times \Sigma \times Q$ is a set of transitions between states. Each element $(src, label, dst) \in T$ designates a source state,src, a $label$, and a destination state, dst.

The finite automaton is deterministic if

$$\forall (s, \ell_1, d_1), (s, \ell_2, d_2) \in T, (d_1 \neq d_2 \implies \ell_1 \neq \ell_2),$$

meaning that distinct transitions from the same source state, s, have different labels, ℓ_1, ℓ_2.

Determinism is violated if $\exists (s, \ell) \in Q \times \Sigma$ such that $(s, \ell_1, d_1), (s, \ell_2, d_2) \in T$ but $d_1 \neq d_2$.

6.7.1 RE $0 + ((1 + 2) \cdot (0 + 1 + 2)^*)$

To illustrate this representation, we look at a subgraph of the DFA shown in Figure 6.2 [Bottom]. The DFA shown Figure 6.4 recognizes all positive base-3 integers.

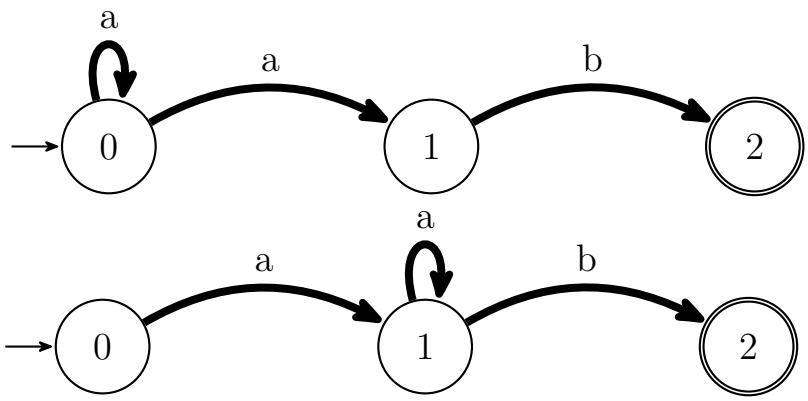


Figure 6.5: [Top] NFA A_1, recognizing the regular expression a^*ab. This finite automaton is non-deterministic, because state 0 has two transitions with the same label. [Bottom] DFA A_2, recognizing the regular expression a^*ab. This finite automaton is deterministic, because no state has two transitions with the same label.

$$
\begin{aligned}
A &= (\Sigma, Q, I, F, T) \\
\Sigma &= \{0, 1, 2\} \\
Q &= \{0, 2, 3\} \\
I &= \{0\} \\
F &= \{2, 3\} \\
T &= \{(0, 0, 2), (0, 1, 3), (0, 2, 3), \\
&\qquad (3, 0, 3), (3, 1, 3), (3, 2, 3)\}
\end{aligned}
$$

6.7.2 RE a^*ab

As a second example, we demonstrate deterministic and non-deterministic finite automata which recognize the regular expression a^*ab, assuming the alphabet $\Sigma = \{a, b\}$.

Figure 6.5 [Top] shows a non-deterministic finite automaton (NFA) implementing a^*ab.

We represent this mathematically and programmatically as $A_1 = (\Sigma, Q, I, F, T)$, where

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{0, 1, 2\} \\
I &= \{0\} \\
F &= \{2\} \\
T &= \{(0, a, 0),\ (0, a, 1),\ (1, b, 2)\}
\end{aligned}
$$

We see that A_1 is non-deterministic, because T contains two transitions $(0, a, 0)$ and $(0, a, 1)$ which have the same source and the same label `"a"`.

As we previously stated, every NFA can be converted to an equivalent DFA. Figure 6.5 [Bottom] shows a DFA implementing the same regular expression.

We can represent this programmatically as $A_2 = (\Sigma, Q, I, F, T)$, where

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{0, 1, 2\} \\
I &= \{0\} \\
F &= \{2\} \\
T &= \{(0, a, 1),\ (1, a, 1),\ (1, b, 2)\}
\end{aligned}
$$

In contrast to A_1 above, A_2 has transitions which are deterministic; *i.e.* the same label does not appear on two different transitions from the same source. The transitions with 0 as source are $\{(0, "a", 1)\}$; the transitions with 1 as source are $\{(1, "a", 1), (1, "b", 2)\}$, and the transitions with 2 as source are $\emptyset$.

6.7.3 Implementing RE $(a^*bb)^*$

The DFA in Figure 6.6 implements the regular expression $(a^*bb)^*$. To convince yourself of this you should test the first several words in the enumerated language: ε, *bb*, *abb*, *aabb*, *bbbb*, *aaabb*, *abbbb*, *bbabb*, *aaaabb*, *aabbbb*, *abbabb*, *bbaabb*,....

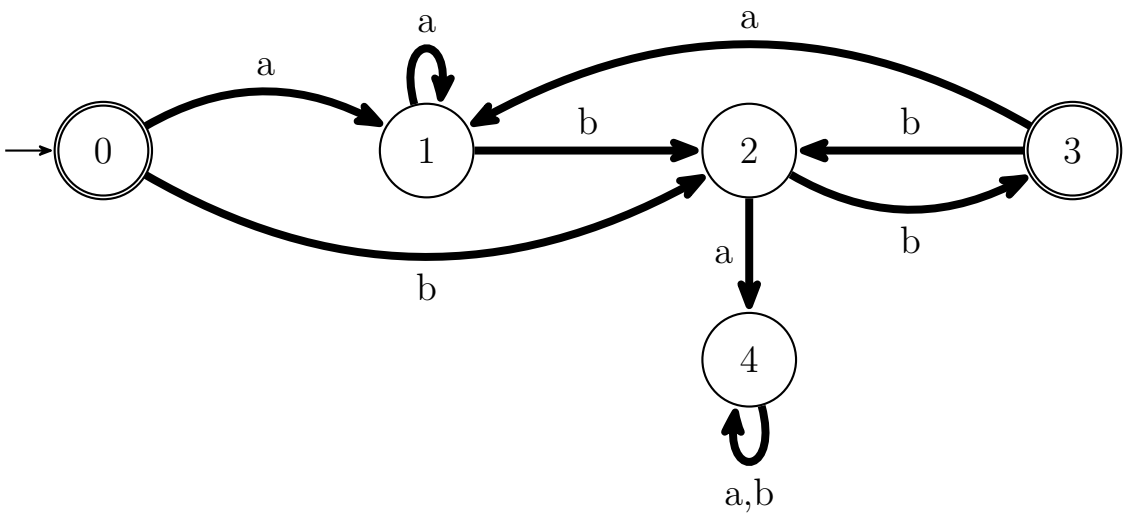


Figure 6.6: DFA A_3, recognizing the regular expression $(a^*bb)^*$.

We can represent this programmatically as $A_3 = (\Sigma, Q, I, F, T)$, where

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{0, 1, 2, 3, 4\} \\
I &= \{0\} \\
F &= \{0, 3\} \\
T &= \{(0, a, 1),\ (0, b, 2), \\
&\qquad (1, a, 1),\ (1, b, 2), \\
&\qquad (2, a, 4),\ (2, b, 3), \\
&\qquad (3, a, 1),\ (3, b, 2), \\
&\qquad (4, a, 1),\ (4, b, 2)\}
\end{aligned}
$$

Note that $0 \in F$ because the empty word matches the regular expression; *i.e.*, $\varepsilon \in [\![(a^*bb)^*]\!]$.

6.8 Regular Expression Matching

We will not discuss how to compute A_1 nor A_2 from the regular expression. However, suppose you are already given A_1 and A_2, we would

like to write a Python function which will decide whether a given string matches the corresponding regular expression. The code can be made more efficient if we know the finite auto is deterministic.

First we develope the functions `dfa_advance_state` and `nfa_advance_state`. This function takes an *current* state and a label, and returns the next state (or None) in the case a DFA, or the set of next states in the case of a NFA.

Given the `dfa_advance_state` (or `nfa_advance_state`) function we can iterate through an input string to find the final state (or set of final states). If one of the final states of this iteration is an accepting state, then we have determined that the string matches the regular expression.

6.8.1 Computing next state of a DFA

If the finite automaton is determistic, the match query can be done in linear time with respect to the string being queried. It is somewhat surprising, but nevertheless true, that the complexity of the regular expression does not contribute to the compute-time complexity of the decisions function.

The trick to get the best performance is to convert the set of transitions, T into a dictionary of dictionaries. In the case of A_2 above, $T = \{(0, "a", 1),\ (1, "a", 1),\ (1, "b", 2)\}$ should be converted to the following dictionary which maps each source state to another (inner) dictionary. The inner dictionary maps each label to destination state.

```
T_as_dict = {0 : {"a" : 1},
             1 : {"a" : 1,
                  "b" : 2},
             2 : {}
}

T_as_dict[0] # returns {"a" : 1}
T_as_dict[0]["a"] # return 1 because (0, "a", 1) is in T
T_as_dict[0]["b"] # throws an error as "b" is not a key in {"a" : 1}
T_as_dict[0].get("b", None) # returns None

T_as_dict[1].get("a", None) # returns 1
T_as_dict[1].get("b", None) # returns 2

T_as_dict[2].get("a", None) # returns None
T_as_dict[2].get("b", None) # returns None
```

The Python function `dfa_advance_state` can be written as follows assuming the set of transitions has been converted to a dictionary of dictionaries.

```
#  Returns either None or the unique next state, d, for which
#   (src, label,d) is in the transitions set.
#   The 'transitions' parameter a dictionary of dictionaries
#   as shown above as T_as_dict.
def dfa_advance_state(transitions, src, label):
  return transitions[src].get(label, None)
```

If optimum performance is not needed, the transitions can be stored simply as a set of 3-tuples, in which case `dfa_advance_state` must search through the transitions each time. The time complexity of this function is linear in `len(transitions)`. This is good for a DFA with few transitions.

```
def dfa_advance_state(transitions, src, label):
  return next((d for s, l, d in transitions
                 if s == src
                 if l == label),
              None)
```

Another possibility is the following. This implementaion is interesting because it does a maximum of `len(Q)` many set membership queries. This is good for a DFA with few states.

```
def dfa_advance_state(transitions, Q, src, label):
  return next((d for d in Q
                 if (src, label, d) in transitions),
              None)
```

6.8.2 Computing next states of an NFA

If the finite automaton is non-determistic, then each state may have multiple transitions for a given label. Therefore, the `nfa_advance_state` function must return some sort of possibly empty container of states, *e.g.* a list or set.

Take as example A_2 shown in Figure 6.5 [Bottom]. Again we can create a dictionary representation of the transitions set.

Recall, by Python notation, `{}` is an empty dictionary, not an empty set. An empty set is created as `set()`.

```
T_as_dict = {0 : {"a" : {1}},
             1 : {"a" : {1,2},
                  "b" : {2}},
             2 : {}
}

T_as_dict[0] # returns {"a" : {1}}
T_as_dict[0].get("b", set()) # returns empty set

T_as_dict[1].get("a", set()) # returns set {1,2}
T_as_dict[1].get("b", set()) # returns set {2}

T_as_dict[2].get("a", set()) # returns empty set
T_as_dict[2].get("b", set()) # returns empty set
```

The Python function `nfa_advance_state` can be written as follows assuming the set of transitions has been converted to a dictionary of dictionaries.

```
#  Returns a possibly empty set of destination states, d, for which
#   (src,label,d) is in the transitions set.
#   The 'transitions' parameter a dictionary of dictionaries as shown
#   above as T_as_dict.
def nfa_advance_state(transitions, src, label):

  return transitions[src].get(label, set())
```

As was the case in Section 6.8.1, another possibility is the following. Given `transitions` as a set of 3-tuples, the following implementation of `nfa_advance_state` returns the list of next states, from a given state and a given label. The set of next states might be the empty set. This implementaion is interesting because it does `len(Q)` many set membership queries. This is good for a DFA with few states.

```
def nfa_advance_state(transitions, Q, src, label):
  return {d for d in Q
            if (src, label, d) in transitions}
```

6.8.3 RE Matching

If the finite automaton is determistic, then walking through it is easy. We just start with q as the initial state, given in `I`. For each character in the given `string`, we compute advance the state according to that character as the transition label. Each time, we must check whether

`advance_state` returned `None`, in which case we abort the iteration and simply return `False`. Otherwise, if we reach the end of the iteration and the value of the advanced state is an element of the list of final states, `F`, then we return `True`, otherwise `False`.

```
def dfa_match(Q, I, F, transitions, string) -> bool:
  assert len(I) == 1
  q = list(I)[0]
  for c in string:
    q = dfa_advance_state(transitions, Q, q, c)
    if q is None:
      return False
  return q in F
```

If the finte automaton is non-determistic, then walking through it is a bit more difficult. In this case `nfa_advance_state` returns a set of next-states, and we must walk though each of them. This, in the worst case, the complexity may be polynomial. *I.e.*, in worst case `qs` might become equal to `Q`, so that every iteration of the `for` set-comprehension calls `nfa_advance_state` `len(Q)`-many times. Assuming the each call to `nfa_advance_state` iterates `for q in Q`, then we have $n \times |Q|^2$ complexity where n is the length of the string being checked and $|Q|$ is the size of `Q`.

```
def nfa_match(Q, I, F, transitions, string) -> bool:
  qs = I
  for c in string:
    qs = {s for q in qs
            for s in nfa_advance_state(transitions, Q, q, c)}
  return any(q in F
             for q in qs)
```

6.8.4 The Fold Pattern

There is a common pattern in programming called *folding*. The process is as follows.

1. Start with some initial value z. This may be a number, set, list, or any other type of object depending on the application.

2. Given some sequence of input values, take the first value, i, from the sequence and call a given function f given the two values $f(z, i)$

to produce a new value with the same type as z.

3. Continue iterating through the sequence, each time calling f. But each time f is called, its first argument is the previous value returned from f.

This process imitates a depth-n composition of the function f. For example, suppose we have a list $[n_0, n_1, n_2, n_3, n_4, n_5] = [2, 5, -6, 2, 0, 8]$ of integers to add, and that f is a binary function which adds is arguments. Then we can add the list as follows:

$$\begin{aligned} f(f(f(f(f(f(0,n_0),n_1),n_2),n_3),n_4),n_5) = \\ f(f(f(f(f(f(0,2),5),-6),2),0),8) = \\ f(f(f(f(f(2,5),-6),2),0),8) = \\ f(f(f(f(7,-6),2),0),8) = \\ f(f(f(1,2),0),8) = \\ f(f(3,0),8) = \\ f(3,8) = 11 \end{aligned}$$

Such a process could be implemented in Python as follows.

```
1  def fold(f,z,input_list):
2    acc = z
3    for i in input_list:
4      acc = f(acc, i)
5    return acc
```

This `fold` function is provided by the Python library `functools` as is named `reduce` rather than `fold`.

```
1 from functools import reduce
2
3 reduce(f,z,input_list)
```

In fact `reduce` (names `fold` in many other programming languages) can be used in many cases when you have a binary function, which you need to apply to multiple arguments. *E.g.*., you can add two numbers, but you need to add multiple numbers; you can add two matrices, but you need to add multiple matrices; you can compute the minimum of two numbers, but you need the minimum of a list of numbers; you can multiply two polynomials, but you need to multiply a list of polynomials.

For example, Suppose you are given a list of polynomials `polys`, and a function which multiplies two polynomials `poly_mult`, and a representation the `one` polynomial (*e.g.*, `[1]`). You may easily multiply the list using `reduce(poly_mult, [1], polys)`.

6.8.5 Using `reduce` with NFA matching

This *folding* pattern is very similar to that which we see in regular expression matching using an NFA. We start with z equal to the set of initial states of the NFA, I. Then we iterate an input string, whose language membership we are trying to query. With each iteration we update the starting states to a new value by calling the function `nfa_advance_state`.

The problem similar but not exact. There are two issues

1. that `nfa_advance_state` takes as 3rd argument, a single state rather than a set of states.

2. that `reduce` needs a binary function as input, but `nfa_advance_state` takes 4 arguments.

We can make the NFA matching problem *fold-compatible* by writing a wrapper function around `nfa_advance_state`.

```
def nfa_advance_states(states, transitions, Q, c):
  return {s for q in states
            for s in nfa_advance_state(transitions, Q, q, c)}
```

Now we can rewrite `nfa_match` using `reduce`.

```
def nfa_match(Q, I, F, transitions, string) -> bool:
  qs = reduce(lambda qs, c: nfa_advance_states(qs, transitions, Q, c),
                                        string, I)
  return any(q in F for q in qs)
```

The solution is slightly more elegant using a local function, because the local function sees the variables, `Q` and `transitions` in scope, eliminating the need for the use of `lambda`.

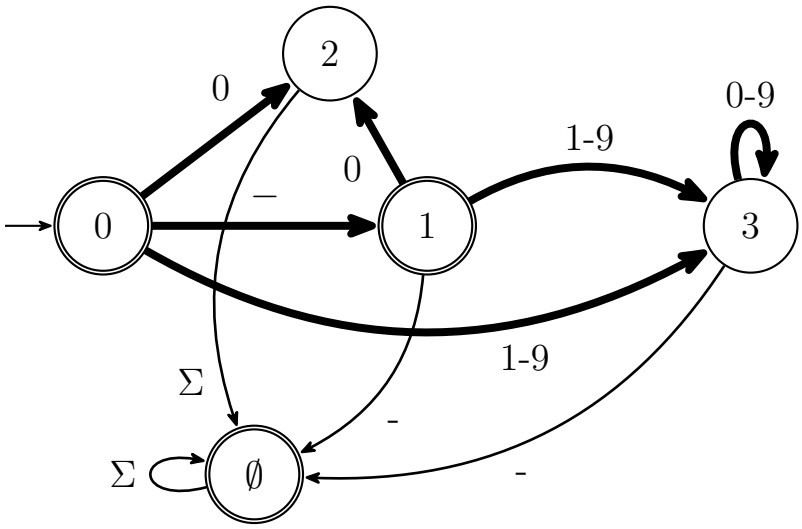


Figure 6.7: Complement DFA, rejecting all strings which represent valid integers, but considering only letters `"-"` and digits 0-9.

```
def nfa_match(Q, I, F, transitions, string) -> bool:
  def nfa_advance_states(states, c):
    return {s for q in states
              for s in nfa_advance_state(transitions, Q, q, c)}

  qs = reduce(nfa_advance_states, string, I)
  return any(q in F
             for q in qs)
```

6.9 Complement of a Regular Expression

Given a regular expression, how can we recognize all strings which fail to match the regular expression. In this section we won't find the regular expression but it is easy to find the DFA representing it. There are well-known algorithms for extracting the regular expression from a DFA, but we will not present that in this course.

The process of complementing a complete DFA is simple. So if the DFA is incomplete, you must first complete it as described in Section 6.6. Now, given a complete DFA $A = (\Sigma, Q, I, F, T)$ the complement DFA is simply $\overline{A} = (\Sigma, Q, I, Q \setminus F, T)$. *I.e.*, we construct a DFA with exactly the same alphabet, same states, same initial state, and same transitions, but we complement the set of accepting states. If q is an accepting state

in A then it becomes non-accepting in $\overline{A}$, and if q is non-accepting in A, it becomes accepting in $\overline{A}$. But be careful, this only works if the DFA is complete, because the sink-state of A must be an accepting state in $\overline{A}$.

Figure 6.7 shows the complement of the DFA from Figure 6.3.

Let's find the complement DFA for, A_2, the DFA shown in Figure 6.5 [Bottom]. First, we recognizing that Figure 6.5 [Bottom] is not complete, so we complete it in Figure 6.8. Next we simply redraw Figure 6.8 [Top] but complement the set of accepting states. A_2 has accepting states $F = \{2\}$, so $\overline{A_2}$ must have accepting states $F = Q \setminus \{2\} = \{0, 1, 3\}$.

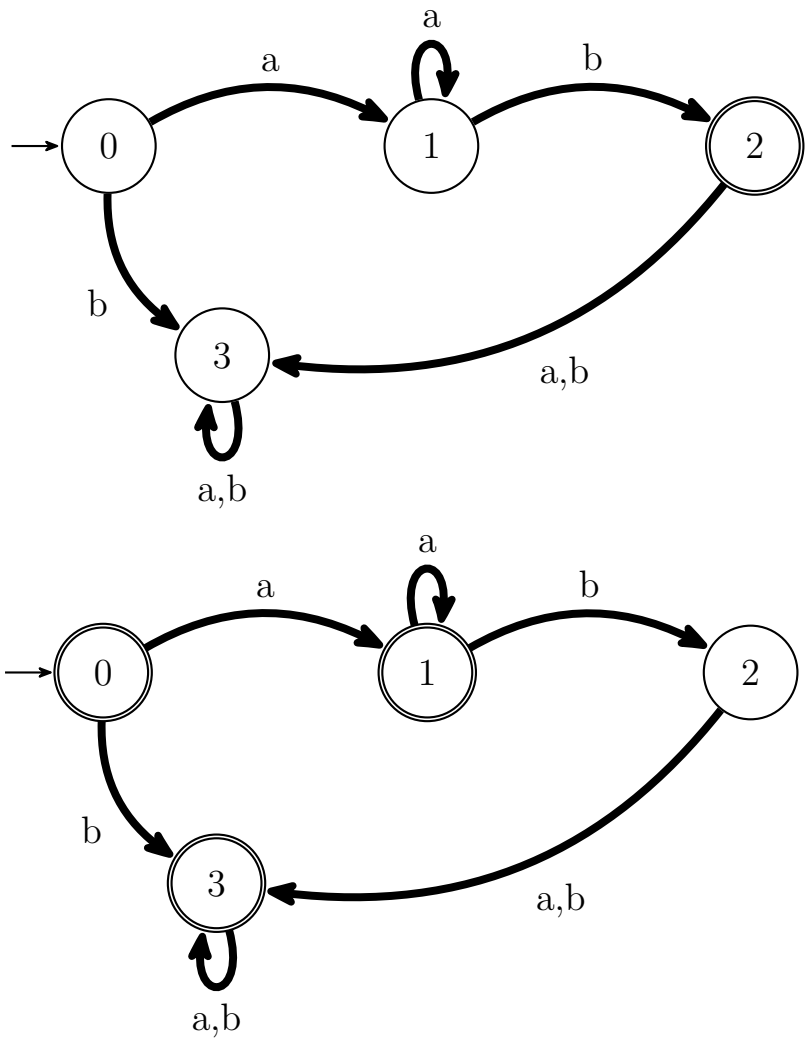


Figure 6.8: Complete version of A_2 from Figure 6.5. [Top] is A_2 completed, [Bottom] is its complement $\overline{A_2}$ with complemented set of accepting states.

Chapter 7

Graph Isomorphism

Chapter Objectives

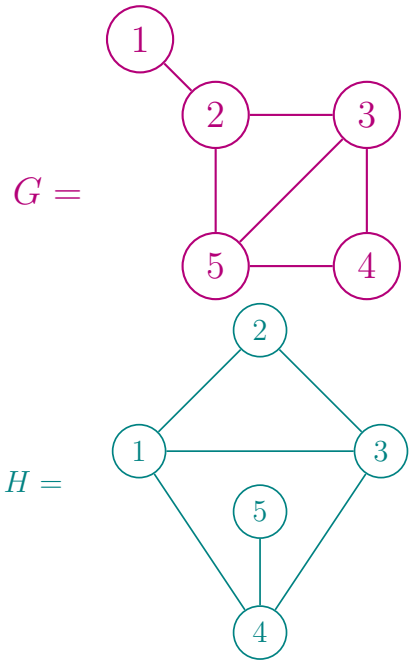
- Represent a graph as an adjacency matrix; space $\Theta(|V|^2)$, or $\Theta(|V| + |E|)$ sparse
- Read k-step walk counts between vertices from A^k
- Verify isomorphism: find permutation matrix P s.t. $PA_HP^{-1} = A_G$
- Extract a vertex bijection from P and express it in cycle notation
- Fuse vertices using a non-square similarity transform PA_GP^T
- Understand zero-knowledge proofs: prove knowledge of h without revealing it

7.1 Adjacency Matrix

The following example is taken from Sandra Herke, in her YouTube video *Graph Theory FAQ03, Isomorphisms using Adjacency Matrix*.

Definition 7.1.1 (*adjacency matrix*)

The *adjacency matrix* of a graph G is the matrix with 1 at row i column j whenever G has an edge connecting vertices v_i and v_j, and



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 7.1: Isomorphic graphs, G and H, and their adjacency matrices

0 otherwise.

$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \qquad A_H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

If graph G is undirected, then whenever v_i connects to v_j, so does v_j connect to v_i. Therefore, for an undirected graph the adjacency matrix is symmetric because entry $A_{i,j} = A_{j,i}$. However, if the graph is directed, we no longer have symmetry.

7.2 Adjacency Matrix and Isomorphism

G and H are isomorphic if and only if there exists an orthogonal matrix P_{HG} such that

$$P_{HG} A_H P_{HG}^{-1} = A_G \,.$$

In such a case P_{HG} is called a *permutation matrix* or the *permutation matrix* going from H to G.

Definition 7.2.1 (*orthogonal matrix*)

An *orthogonal matrix*, M, has orthogonal unit vectors as rows and columns and has the special property that

$$M^{-1} = M^T$$

its inverse is identical to its transpose.

So to calculate the inverse of P_{HG}, we simply transpose it.
Finding P_{HG} is difficult, but verifying it is straightforward.

$$P_{HG} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Let's check whether $P_{HG}A_H P_{HG}^T = A_G$.

$$P_{HG}A_H P_{HG}^T = \overbrace{\begin{bmatrix}0&0&0&0&1\\0&0&0&1&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\end{bmatrix}\begin{bmatrix}0&1&1&1&0\\1&0&1&0&0\\1&1&0&1&0\\1&0&1&0&1\\0&0&0&1&0\end{bmatrix}}^{\text{Associative, so we can multiply these first.}}\begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}$$

$$= \begin{bmatrix}0&0&0&1&0\\1&0&1&0&1\\0&1&1&1&0\\1&0&1&0&0\\1&1&0&1&0\end{bmatrix}\begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}$$

$$= \begin{bmatrix}0&1&0&0&0\\1&0&1&0&1\\0&1&0&1&1\\0&0&1&0&1\\0&1&1&1&0\end{bmatrix} = A_G$$

In addition, given $P_{HG} = \begin{bmatrix}0&0&0&0&1\\0&0&0&1&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\end{bmatrix}$, we can *extract* the isomorphism. To do so, we note that any 1 in row i column j means that vertex v_j of graph H maps to vertex v_i of graph G.

	col		row
$(P_{HG})_{i,j}$	A_H	$\to$	A_G
$(P_{HG})_{1,5}$	5	$\to$	1
$(P_{HG})_{2,4}$	4	$\to$	2
$(P_{HG})_{3,1}$	1	$\to$	3
$(P_{HG})_{4,2}$	2	$\to$	4
$(P_{HG})_{5,3}$	3	$\to$	5

Looking at P_{HG} we conclude that $1_H \to 3_G$, $3_H \to 5_G$, $5_H \to 1_G$, $2_H \to 4_G$, and $4_H \to 2_G$.

Provided the set of vertices of G and H are the same, $V = \{1, 2, 3, 4, 5\}$, in our case, we can represent an isomorphism as permutation, which can be denoted in so-called *cycle notation*. Notice that $1 \to 5 \to 3 \to 1$ is a cycle and so is $2 \to 4 \to 2$. This permutation in terms of its constituent cycles is denoted $A_{HG} = (1\ 3\ 5)(2\ 4)$.

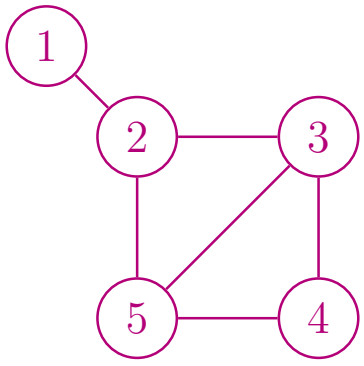
Note that an alternate permutation would be $A_{HG} = (1\ 5)(3)(2\ 4)$, because vertices 1 and 3 in graph H (corresponding to 3 and 5 in graph G) are indistinguishable.

To summarize, the adjacency matrix can be used to *verify* a suspected isomorphism. Given either the permutation cycles or the permutation matrix, we can easily calculate the other, but to calculate either of them given only the graphs is difficult.

7.3 Adjacency Matrix and k-step Walks

Another use of the adjacency matrix is to count the number of k-step walks between two given vertices.

If A is the adjacency matrix, then $(A^k)_{i,j} = (A^k)_{j,i}$ is precisely the number of k-step walks from vertex v_i to vertex v_j, and consequently from vertex v_j to vertex v_i.

Let's look again at $G =$ 

.

$$A_G{}^0 = A_G \cdot A_G{}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_G{}^1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

The two trivial cases: from $A_G{}^0$, the identity matrix, we see that there is one 0-step walk from every vertex to itself, and from $A_G{}^1$ we see that there is exactly one 1-step walk from every vertex v_i to each vertex v_j if and only if $\{v_i, v_j\}$ is an edge of G.

$$A_G{}^2 = A_G \cdot A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix}$$

From $A_G{}^2$ we see something more interesting. We see 3's in at $(2, 2)$, $(3, 3)$, and $(5, 5)$; thus there are exactly 3 2-step walks from v_2 to itself, namely

1. $v_2 \to v_1 \to v_2$,
2. $v_2 \to v_3 \to v_2$, and
3. $v_2 \to v_5 \to v_2$;

from v_3 to itself, namely

1. $v_3 \to v_2 \to v_3$,
2. $v_3 \to v_4 \to v_3$, and

3. $v_3 \to v_5 \to v_3$;

and from v_5 to itself, namely

1. $v_5 \to v_2 \to v_5$,
2. $v_5 \to v_3 \to v_5$, and
3. $v_5 \to v_4 \to v_5$.

Likewise, there is a 2 at row 4 column 2 (and at row 2 column 4) meaning that there are 2 2-step walks from vertex v_2 to vertex v_4, namely

1. $v_2 \to v_3 \to v_4$, and
2. $v_2 \to v_5 \to v_4$.

$$A_G{}^3 = A_G{}^2 \cdot A_G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 1 & 2 & 1 \\ 3 & 2 & 6 & 2 & 6 \\ 1 & 6 & 4 & 5 & 5 \\ 2 & 2 & 5 & 2 & 5 \\ 1 & 6 & 5 & 5 & 4 \end{bmatrix}$$

The calculation of $A_G{}^3$ predicts that there are 6 3-step walks between v_2 and v_5, namely

1. $v_2 \to v_1 \to v_2 \to v_5$,
2. $v_2 \to v_3 \to v_2 \to v_5$,
3. $v_2 \to v_5 \to v_2 \to v_5$,
4. $v_2 \to v_3 \to v_4 \to v_5$,
5. $v_2 \to v_5 \to v_3 \to v_5$, and
6. $v_2 \to v_5 \to v_4 \to v_5$.

Space requirement for the adjacency matrix is $\Theta(|V|^2)$, sometimes written $\Theta(V^2)$. This can become $\Theta(|V| + |E|)$ if a sparse matrix is used.

An advantage of the adjacency matrix which we have not yet mentioned is that given A_G, we can easily determine whether v_i connects to v_j in constant time, and we can easily determine the list of connecting vertices of v_i in linear time.

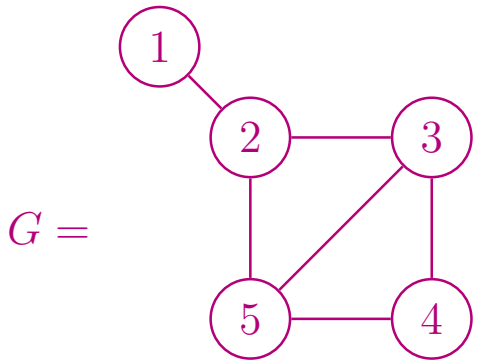


Figure 7.2: We would like to fuse vertex 2 with vertex 5

7.4 Fusing Vertices

The fusion of vertices in a graph can also be represented as a similarity transformation, but of course not an invertible transformation. Imagine we wish to fuse vertices 2 and 5 in the graph G as shown in Figure 7.2. The new graph, which we will called G_{fused} will be represented by an adjacency matrix named A_{fused}.

The adjacency matrix of A_{fused} is the following 4×4 matrix. But how can we compute A_{fused} given A_G? The answer is using a similarity transformation $A_{fused} = PAP^T$ where P is necessarily a 4×5 matrix, and P^T is a 5×4 matrix.

In fact to construct P with the intent of fusing vertices i and j, we start with the $n \times n$ identity matrix, where $n \times n$ is the dimension A_G. We remove row j, and replace row i with the sum of rows i and j—otherwise stated, row j becomes a row of zeros except ones in columns i and j. This has the added effect of renumbering all rows greater than j.

For our example we with to merge vertices 2 and 5, so we can remove row 5 of the 5×5 identity matrix and replace row 2 with

$$\begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0 & 1\end{bmatrix}.$$

Now, we can compute A_{fused} as follows.

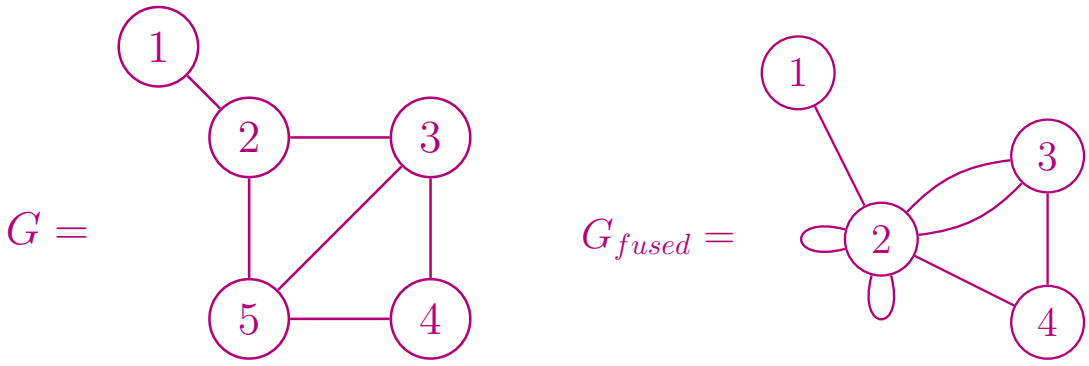


Figure 7.3: Undirected graph G and the result G_{fused} after fusing vertices 2 and 5.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A_{fused} = P\ A_G\ P^T$$

$$= \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}}_{A_G} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{P^T}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 2 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

An initial reaction looking at A_{fused} and at the graph G_{fused} in Figure 7.3 is that there must be a mistake. Why are there two self-loops connected to vertex 2, and why is there a 2 in the 2nd row 2nd column of the matrix? This is not a mistake. The 2 (and the double self-loop) distinguishes the cases of fusing vertices in a directed graph vs an undirected graph. If G had been a directed graph, then a single-directed edge from 2 to 5, would have fused into a single self-loop and a 1 in the matrix.

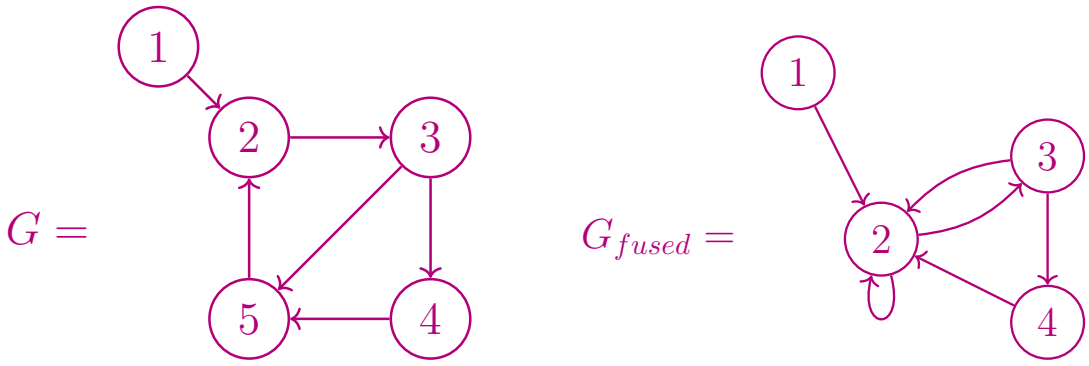


Figure 7.4: We would like to fuse vertex 2 with vertex 5, of the directed graph.

But since the edge in our undirected graph G is bidirectional, two self loops result and a 2 in the matrix results.

Figure 7.4 shows an example similar to Figure 7.2 except with directed edges. In this case we have a non-symmetric A_G; but P is the same as before.

$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

So we again compute the adjacency matrix.

$$\begin{aligned} A_{fused} &= P\ A_G\ P^T \\ &= \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}}_{A_G} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{P^T} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{aligned}$$

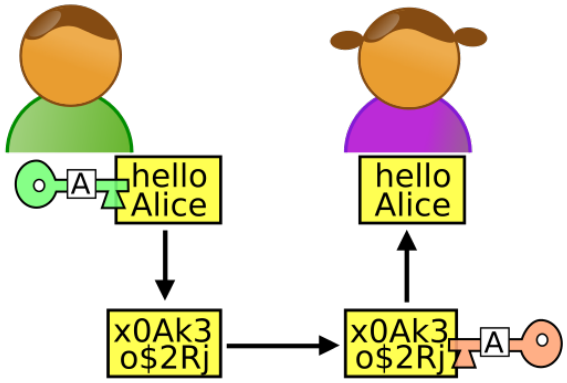


Figure 7.5: Alice wants to prove to Bob that she knows the secret key.

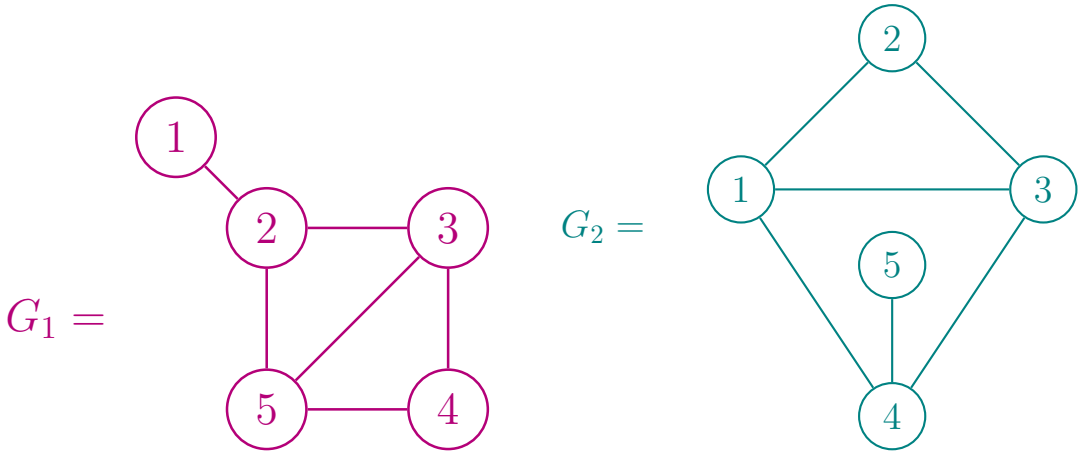


Figure 7.6: Public graphs G_1 and G_2

7.5 Zero Knowledge Proofs, ZKPs

A zero-knowledge proof is a demonstration that someone knows a secret, but without revealing the secret, nor without revealing enough information to allow someone to reconstruct the secret. Neither can the person verifying the validity reconstruct the secret, nor can anyone eavesdropping reconstruct the secret.

The example we will look at is the following. Imagine that two very large isomorphic graphs are publicly available, but that the isomorphism between them is secret. How can someone (Alice) prove she knows this isomorphism, but without allowing Bob (or anyone else) to reconstruct the isomorphism?

7.5.1 Graph Isomorphism ZKP

Consider that two graphs G_1 and G_2 as in Figure 7.6 are publicly known. If Alice knows the isomorphism $h = (5\ 3\ 1)(2\ 4)$, but Bob does not. Alice would like to convince Bob that she knows the isomorphism. Recall that it is difficult for Bob to compute an isomorphism from scratch, but easy to verify.

Question: Can Alice convince Bob that she knows h without revealing h to Bob nor allowing Bob to reconstruct h?

Answer: Yes! Here is how.

1. Alice chooses, μ, a secret random permutation of $[1, 2, 3, 4, 5]$. *E.g.*, $\mu = (1\ 2)(5\ 3\ 4)$; see Figure 7.7.

2. Alice randomly chooses G_1 or G_2 from Figure 7.6. *I.e.*, $a \in \{1, 2\}$.

3. Alice computes $H = \mu G_a$; see Figure 7.7. *I.e.*, either $H = \mu G_1 = G_1'$ or $H = \mu G_2 = G_2'$: Alice knows; Bob doesn't.

4. Alice sends H to Bob, but neither μ nor a. Bob does not know whether he has $H = G_1'$ or $H = G_2'$; neither does the eavesdropper. Bob has $H \in \{\mu G_1, \mu G_2\} = \{G_1', G_2'\}$, but he doesn't know which.

5. Bob randomly chooses G_1 or G_2. *I.e.*, $b \in \{1, 2\}$.

6. Bob sends b to Alice. Bob and Alice both know b, and perhaps the eavesdropper.

7. Bob challenges Alice to produce λ such that $\lambda H = G_b$. *I.e.*, Bob asks for the isomorphism from H to G_1 or G_2. He will easily be able to verify the isomorphism, as that is easy.

8. Alice computes λ, and sends it to Bob.

9. Bob verifies that $\lambda H = G_b$

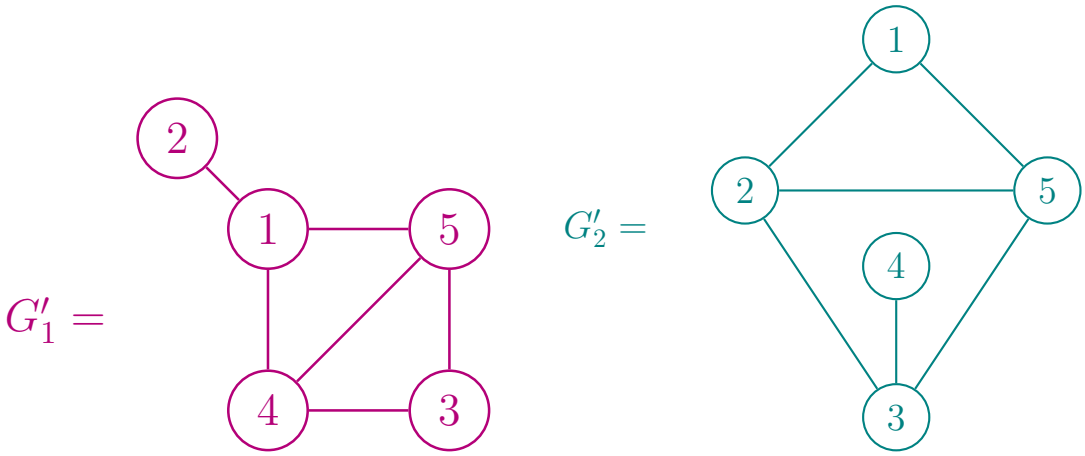
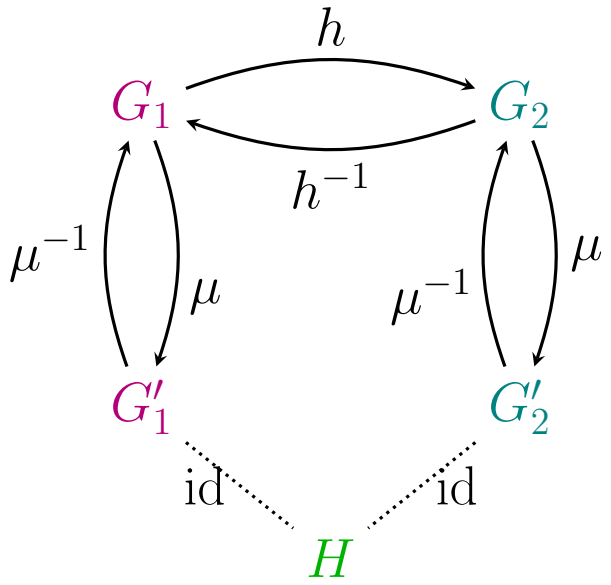


Figure 7.7: Alice computes $\mu = (1\ 2)(5\ 3\ 4)$ and either $H = \mu G_1$ or $H = \mu G_2$.



a	b / G_a'	1 / G_b	2 / G_2
1	$H = G_1'$	μ^{-1}	$h \circ \mu^{-1}$
2	$H = G_2'$	$h^{-1} \circ \mu^{-1}$	μ^{-1}

Figure 7.8: Transformations between pairs of graphs

7.5.2 Does it work?

Should Bob be convinced that Alice knows h, simply because she had demonstrated that she can find λ such that $\lambda H = \lambda G'_a = G_b$?

Bob is not 100% convinced. 

- Bob knows b but not a.
- There is a 50% chance that $a = b$,
- Alice might have sent μ^{-1} without knowing h.
- So Bob must repeat the experiment n times,
- to have a certainty of $1 - \frac{1}{2^n}$.

Chapter 8

Breadth First Search

Chapter Objectives

- Implement BFS with a FIFO queue; analyse complexity as $O(|V|+|E|)$
- Track BFS frontiers $\Phi_0, \Phi_1, \ldots$ to compute shortest-path distances
- Define and compute eccentricity, diameter, radius, center, and periphery
- Prove $diameter(G) \leq 2 \times radius(G)$ via the triangle inequality
- Encode Rubik's cube state as a 64-bit base-6 integer for memory efficiency
- Apply BFS to the cube state graph to find minimum-twist solutions (God's number)

In this chapter we will look at the Breadth First Search (BFS), and we will use the $2 \times 2 \times 2$ Rubik's cube as an example. As we will see, we can examine the Rubik's cube graph using the BFS algorithm.

8.1 Intuition

Figure 8 shows an intuitive look at the DFS (depth first search) vs the BFS (breadth first search). Intuitively DFS and BFS are tree traversals, except that the algorithm works not only for trees but also for general

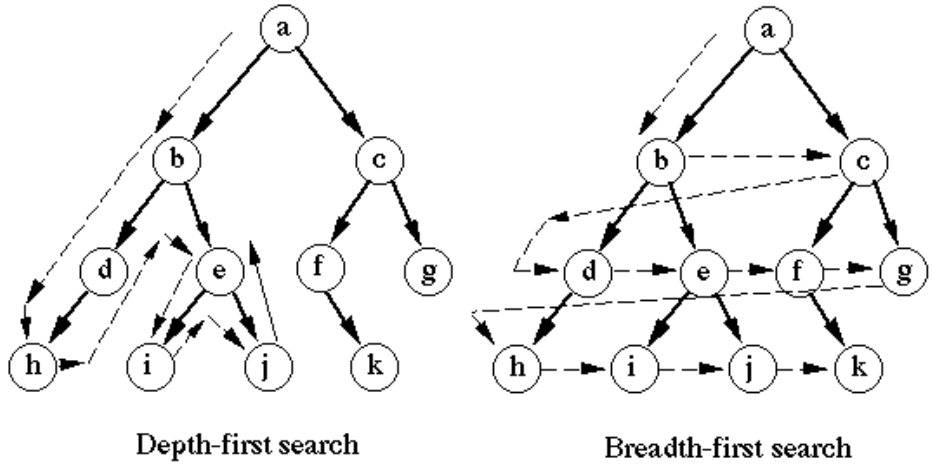


Figure 8.1: Intuition of DFS vs BFS for trees

graphs. A graph may differ from a simple tree as shown in Figure 8.1, in that:

- There may be no distinguished root.
- A vertex may be a neighbor of multiple vertices.
- There may be loops.
- The graph may be disconnected.
- Edges may be directed or undirected.

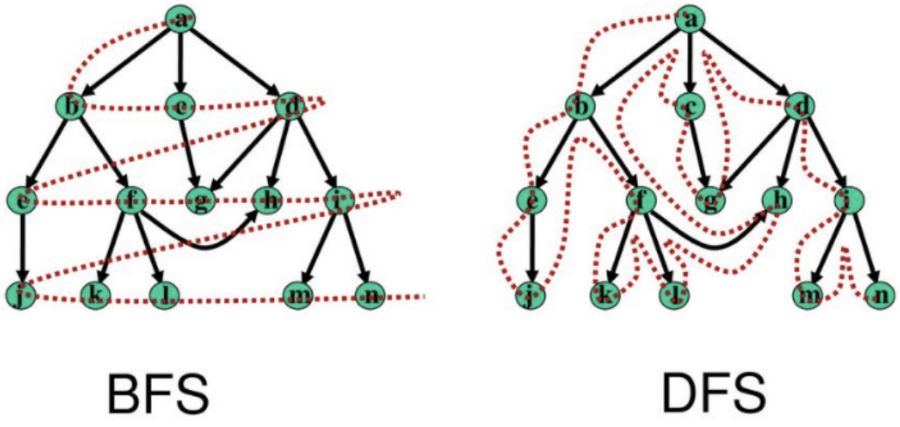


Figure 8.2: DFS vs BFS for graphs that differ from trees.

- The graph may be generated lazily.

Usually, we would like to *visit* each vertex once and only once, and we have to avoid infinite loops. Therefore both algorithms may need some sort of memory data structure to remember which edges have already been visted, and to assure that no vertex is missed.

Algorithm 5 outlines one approach to implementing the BFS, using a so-called *FIFO* (first-in first-out queue).

Algorithm 5: BFS(s,Adj)

Input : s a starting vertex
Input : Adj graph adjacency list, as a mapping

```
1  level[s] ← 0
2  Π[s] ← None
3  fifo.push(s)
4  while fifo not empty do
5  |   v ← fifo.pop()
6  |   for u ∈ Adj[v] do
7  |   |   if not Π[u] then
8  |   |   |   Π[u] ← v
9  |   |   |   level[u] ← 1 + level[v]
10 |   |   |   fifo.push(u)
```

To analyze the complexity, notice that every reachable edge goes into the FIFO exactly once (push), and comes out (pop) exactly once. The `while` loop repeats $|V|$ times. The complexity is

$$O(|V| + |E|)$$

The key insight is that *for* $u \in Adj[v]$ occurs once per edge (or twice for an undirected graph). Recall

$$\sum_{v \in V} degree(v) = 2|E| = \Theta(|E|)$$

As an alternative to the FIFO approach to BFS, Algorithm 6 shows how to use two stacks, one for pushing and one for popping. When a vertex is visited, its children are pushed into the FIFO (unless they have

already been visited). The FIFO data structure assures that all parent vertices are visited before their children.

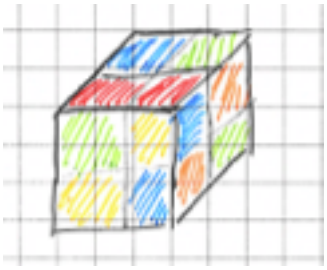
Algorithm 6: BFS(s,Adj)

Input : s a starting vertex
Input : Adj graph adjacency list, as a mapping

```
1  level[s] ← 0
2  eccentricity ← 0
3  Π[s] ← None
4  frontier ← empty stack
5  frontier.push(s)
6  while frontier not empty do
7  |  next ← empty stack
8  |  for v ∈ frontier do
9  |  |  for u ∈ Adj[v] do
10 |  |  |  if not Π[u] then
11 |  |  |  |  next.push(u)
12 |  |  |  |  Π[u] ← v
13 |  |  |  |  level[u] ← eccentricity
14 |  |  frontier ← next
15 |  |  eccentricity = 1 + eccentricity
```

8.2 Rubik's Cube

Let's take a look the $2 \times 2 \times 2$ Rubiks cube.



The cube has 6 faces, each having 4 stickers. There are 4 stickers of each color. So $24 = 6 \times 4$ stickers, or positions. Each position has a color, as denoted in Figure 8.3, but twisting the cube changes which colors are associated with which position.

As shown in Figure 8.4, we can represent a state of the cube as an

Abbreviation	Color
R	red
W	white
B	blue
G	green
O	orange
Y	yellow

Figure 8.3: Color abbreviations

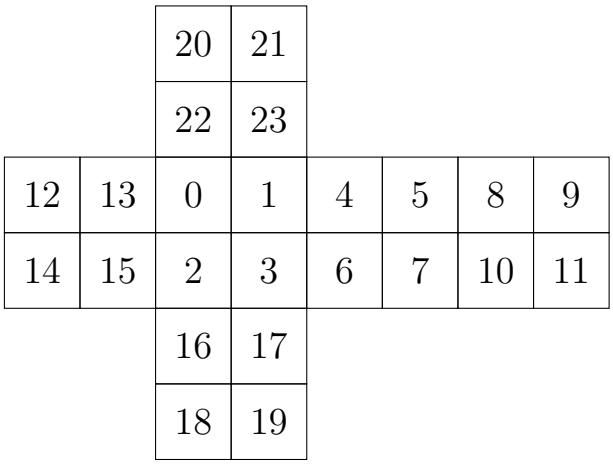


Figure 8.4: Positions on the cube, numbers 0 to 23

array of size 24. So each of the 24 positions has an index, and the value at that index is the color of the sticker at that position.

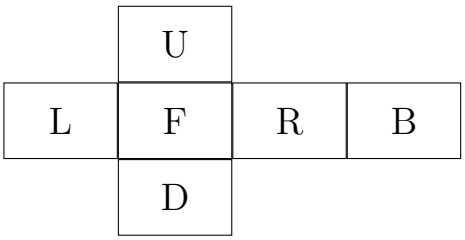
As shown in Figure 8.5, each face has a name. To help with notation, we denote each of the 6 faces by a single letter.

We can represent any state by an appropriate permutation of the string `WWWWGGGGYYYYBBBBOOOORRRR`. However, not all permutations are valid configurations of the cube.

8.3 Size of state space

How many different states can the cube be in; i.e., what is the size of the state space?

One way to bound the size is by asking what the largest 24 digit base 6 integer is. I.e., if we start with any string representing a state, for



Abbreviation	Face
F	front
B	back
U	up
D	down
L	left
R	right

Figure 8.5: The names of each of the 6 faces of the cube

example `GWRGYYRBYWBGOROWORB`, and map it to a number according to the following map,

$$\begin{aligned} W &\to 0 \\ G &\to 1 \\ Y &\to 2 \\ B &\to 3 \\ 0 &\to 4 \\ R &\to 5 \end{aligned}$$

then we arrive at a 24 digit base 6 number such as

$$\text{BGWRGYYRBYWBGOROWORB} \;\to\; 310512253203145404536 \,.$$

The largest 24 digit base six integer is

$$6^{24} - 1 < 4.73838 \times 10^{18} \,.$$

However, the largest 24 digit base 6 integer we can write with exactly four 5's, four 4's, four 3's, four 2's, four 1's, and four 0's is

$$555544443333222211110000_6 < 4.73765 \times 10^{18} \,,$$

which is not much smaller.

This bound is actually excessively high. How many of these 4.73765×10^{18} potential states are actually reachable by rotations

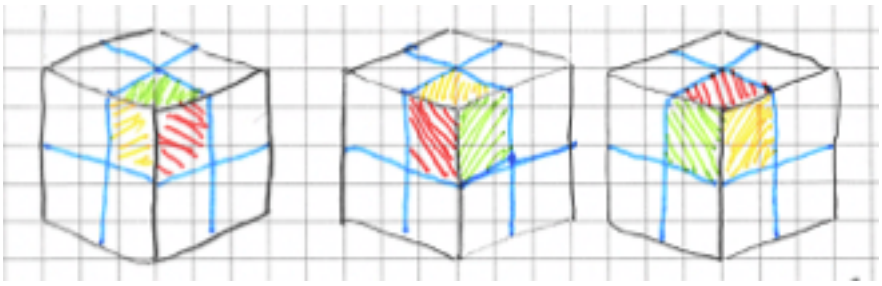


Figure 8.6: Cube consists of 8 cubelets, each with 3 faces

(twists) of the $2 \times 2 \times 2$ Rubik's cube? The cube consists of 8 *cubelets*, each with 3 or stickers or *facelets*. As shown in Figure 8.6, one might imagine any of the cubelets in one of 8 positions, and with any of 3 orientations, $|S| < 8! \times 3^8 = 265,539,520$. We are careful not to claim that all of these states are reachable by rotating the cube–we are merely trying to bound the size of the state space.

Finally, if we realize that we can rotate the entire cube in our hand to look at any face, without actually twisting the cube, we see that we can look at any of 6 faces, and while looking at the face we can turn the entire cube so that any of the adjacent faces is on top. That means there are $6 \times 4 = 24$ orientations of the cube which we can consider isomorphic. So we can further divide $8! \times 3^8$ by 24 to arrive at

$$\frac{8! \times 3^8}{24} = 11,022,480 < 4.73765 \times 10^{18}\,,$$

which is much less than the upper bounds we derived above.

The take-away is that most of what we first considered to be potential states obtainable from permuting the stickers are not actually reachable by twisting the faces of the cube. We will show experimentally, not theoretically, that the actual size of the state space is exactly $\frac{1}{3}$ of this number; $\frac{1}{3} \times \frac{8! \times 3^8}{24} = 3,674,160$.

The $\frac{1}{3}$ is related to the fact that it is impossible to twist one cubelet to any one of its 3 potential orientations without twisting or displacing another–a fact which we will not prove.

The possible rotations (twists) of the cube can be characterized as follows. Each face can be turned either of two directions, 90°

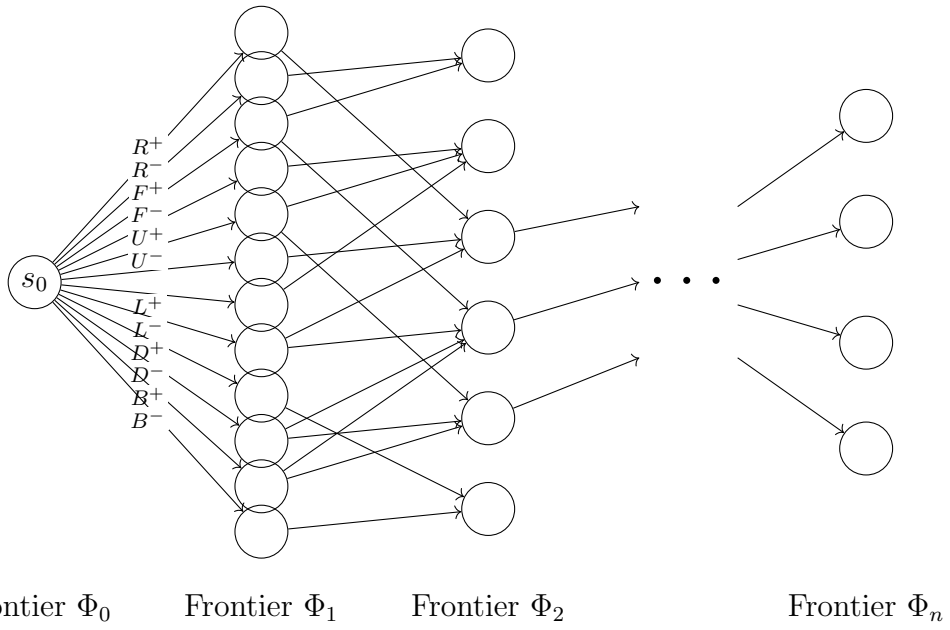


Figure 8.7: Frontiers Φ_0 through Φ_n

F+	F-	front
B+	B-	back
L+	L-	left
R+	R-	right
U+	U-	up
D+	D-	down

We will consider 180° twists (F2, B2, L2, R2, U2, and D2) later.

Let's define a concept called *frontier*. The idea is illustrated in Figure 8.7.

Definition 8.3.1 (*frontier* Φ_0)

The set containing the singleton state of the solved cube, we call this ground state *frontier 0*, denoted Φ_0.

Definition 8.3.2 (*frontier* Φ_1)

Start from the solved state. The set of states reachable from the solved state by exactly one twist, we call *frontier 1*, and denote it Φ_1.

Definition 8.3.3 (*frontier* Φ_2)

The set of all states which are reachable from some state in Φ_1 by exactly one twist, excluding any state already encountered, we call this set of states *frontier 2*, and denote it Φ_2.

For example, if we arrive at *frontier* Φ_1 by turning the cube F+ we can return to *frontier* Φ_0 with the twist F-, because every twist is easily and uniquely reversible.

Definition 8.3.4 (*frontier* Φ_n)

If $n > 0$ we define Φ_n (*frontier* n) as the set of states which are reachable from Φ_{n-1} with a single twist, but are not in $\Phi_0 \cup \Phi_1 \cup ... \cup \Phi_{n-1}$.

We can in this manner bound the size of each frontier. $|\Phi_0| = 12^0 = 1$, exactly 1 state, the solved state. Frontier Φ_1 has exactly $|\Phi_1| = 12^1 = 12$, one for each of the 90° twists. Frontier Φ_2, $|\Phi_2| < 12^2 = 144$. Less, because some of the states were seen in $\Phi_0 \cup \Phi_1$, and because some of the 144 are duplicates. For example, we start in Φ_0 and arrive in Φ_2 by F+ F+, or we can arrive at the exact same state by F- F-.

8.4 God's number

Given two vertices A and B of the graph, a shortest path (maybe not unique), from A to B represents a best sequence of twists to transform the cube from state A to state B.

In particular a shortest path from any vertex v to v_{solved} represents a minimum sequence of twists to solve the cube from that given shuffled state.

What is the maximum such shortest path?

Without 180° twists the answer is 14, and if 180° are allowed, the number is 11. This number is called *God's Number*. It is the number of twists God would need to solve the cube, assuming he always knows the best tactic. God's number for the $3 \times 3 \times 3$ cube was proven in 2010 to be 20. God's number is unknown for any size larger than $3 \times 3 \times 3$.

8.5 Bounds on a graph

Definition 8.5.1 (*path, path length*)

Given a graph, $G = (V, E)$, let P be a sequence of *distinct* edges $(e_1, e_2, \ldots, e_n)$ ($e_i \in E$ and no two e_i are identical), such that for each $1 < i < n - 1$ then $e_i \cap e_{i+1} \neq \emptyset$. Note that $e_1 \setminus e_2$ is a set of exactly one vertex; call it u. And $e_n \setminus e_{n-1}$ is a set of exactly one vertex; call it v. Then P is called a *path* from u to v. The number of elements of P is called the *path length*.

Definition 8.5.2 (*distance*)

Given a graph, $G = (V, E)$, and two vertices, u, and v. There are finitely many paths from u to v. The length of a shortest such path is called the *distance* from u to v, and denoted $\delta(u, v)$.

Definition 8.5.3 (*eccentricity*)

Given a graph, $G = (V, E)$. If $v \in V$, the *eccentricity* of v is the maximum distance from v. Ie.,

$$eccentricity(v) = \max_{u \in V} \delta(v, u) .$$

An example of eccentricities is shown in Figure 8.8.

Definition 8.5.4 (*diameter*)

Given a graph, $G = (V, E)$. The *diameter* of G is defined by

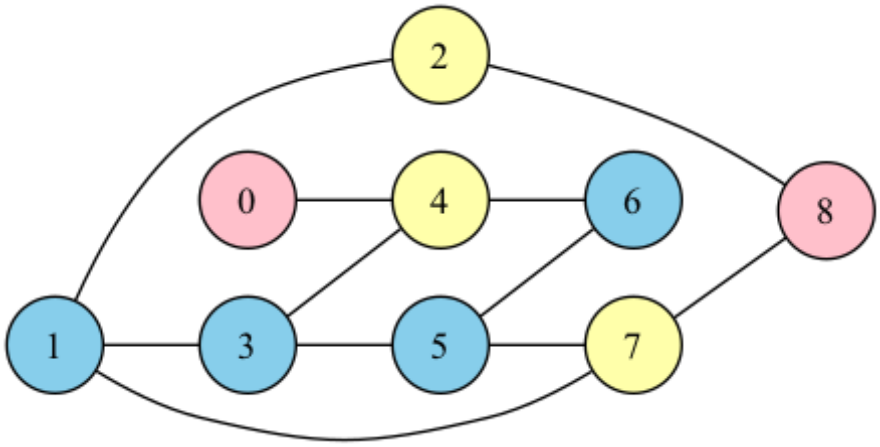
$$diameter(G) = \max_{v \in V} eccentricity(v) .$$

Definition 8.5.5 (*radius*)

Given a graph, $G = (V, E)$. The *radius* of G is defined by

$$radius(G) = \min_{v \in V} eccentricity(v) .$$

Note that by the triangle inequality, the diameter of graph G is bounded by twice the eccentricity of any given vertex.



	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**eccentricity**	
0	0									5	periphery
1	3	0								3	center
2	4	1	0							4	
3	2	1	2	0						3	center
4	1	2	3	1	0					4	
5	3	2	3	1	2	0				3	center
6	2	3	4	2	1	1	0			3	center
7	4	1	2	2	3	1	2	0		4	
8	5	2	1	3	4	2	3	1	0	5	periphery

Figure 8.8: Illustration of periphery, center, and eccentricity

Claim 8.5.6

$$\mathit{diameter}(G) \leq 2 \times \mathit{radius}(G)$$

Proof. Let D be $\mathit{diameter}(G)$, and $R = \mathit{radius}(G)$.
Let $v_1, v_2 \in V$ such that $\delta(v_1, v_2) = D$.
And take $u \in V$ such that $\mathit{eccentricity}(u) = R$.
By the triangle inequality we have:

$$D \leq \delta(v_1, u) + \delta(u, v_2)$$

We know $\delta(v_1, u) \leq R$ and $\delta(u, v_2) \leq R$, because R is the eccentricity of u, every vertex is within a distance R of u.
Thus

$$D \leq \delta(v_1, u) + \delta(u, v_2) \leq R + R = 2R\,.$$

God's number is the eccentricity of the solved state.

Definition 8.5.7 (*periphery*)

Given a graph, $G = (V, E)$, the set $P_G \subset V$ of vertices whose eccentricity equal the diameter is called the *periphery*.

$$P_G = \{v \in V \mid \mathit{eccentricity}(v) = \mathit{diameter}(G)\}$$

A vertex in the periphery of a graph is called a *peripheral vertex*.

An example of peripheral verices is shown in Figure 8.8.

Definition 8.5.8 (*center, central vertex*)

Given a graph, $G = (V, E)$, the set $C_G \subset V$ of vertices whose eccentricity equal the radius is called the *center*.

$$C_G = \{v \in V \mid \mathit{eccentricity}(v) = \mathit{radius}(G)\}$$

A vertex in the center of a graph is called a *central vertex*.

An example of central verices is shown in Figure 8.8.

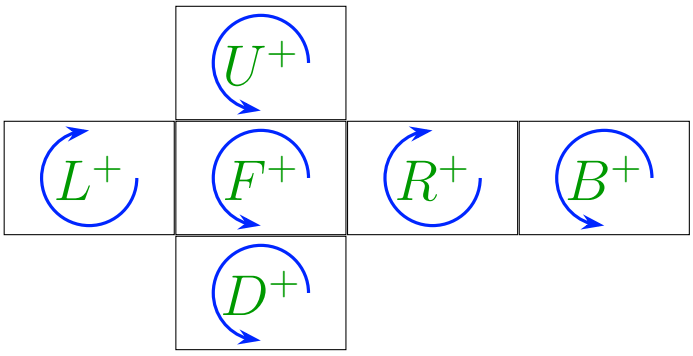


Figure 8.9: Positive twists of each face

8.6 Implementing the Rubik's cube

Recall cycle notation for specifying a permutation. A permutation that transforms (123456) to (421365) can be denoted as (143)(56), meaning $1 \mapsto 4 \mapsto 3 \mapsto 1$ and $5 \mapsto 6 \mapsto 5$. We can represent a twist of the cube in terms of the cycles of its permutation of the facelets, as shown in Figure 8.9.

We can represent each of the forward 90° twists as the following cycles. For example, figure 8.10 illustrates the F+ twist.

F+	=	(0 2 3 1)	(4 22 15 17)	(6 23 13 16)
B+	=	(5 19 14 20)	(7 18 12 21)	(8 10 11 9)
L+	=	(0 16 11 20)	(2 18 9 22)	(12 13 15 14)
R+	=	(1 21 10 17)	(3 23 8 19)	(4 5 7 6)
U+	=	(0 4 8 12)	(1 5 9 13)	(20 22 23 21)
D+	=	(2 14 10 6)	(3 15 11 7)	(16 18 19 17)

The inverse rotations, can be formed by reversing the lists.

F-	=	(1 3 2 0)	(17 15 22 4)	(16 13 23 6)
B-	=	(20 14 19 5)	(21 12 18 7)	(9 11 10 8)
L-	=	(20 11 16 0)	(22 9 18 2)	(14 15 13 12)
R-	=	(17 10 21 1)	(19 8 23 3)	(6 7 5 4)
U-	=	(12 8 4 0)	(13 9 5 1)	(21 23 22 20)
D-	=	(6 10 14 2)	(7 11 15 3)	(17 19 18 16)

The 180° rotations are just double applications of the 90° twists.

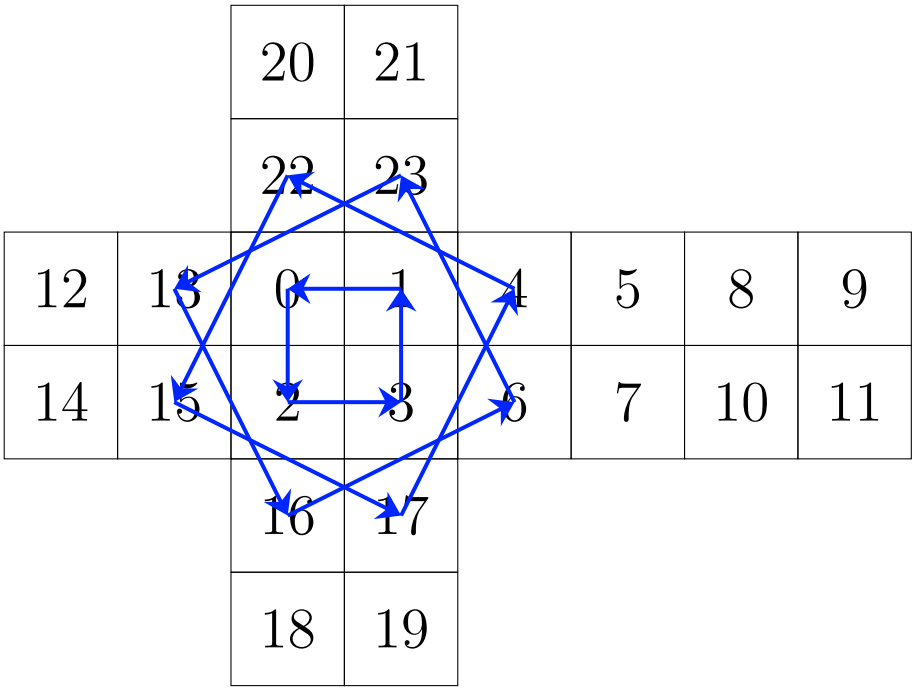


Figure 8.10: Cycles implementing the F+ twist

F2	=	(1 3 2 0)	(17 15 22 4)	(16 13 23 6)
		(1 3 2 0)	(17 15 22 4)	(16 13 23 6)
B2	=	(20 14 19 5)	(21 12 18 7)	(9 11 10 8)
		(20 14 19 5)	(21 12 18 7)	(9 11 10 8)
L2	=	(20 11 16 0)	(22 9 18 2)	(14 15 13 12)
		(20 11 16 0)	(22 9 18 2)	(14 15 13 12)
R2	=	(17 10 21 1)	(19 8 23 3)	(6 7 5 4)
		(17 10 21 1)	(19 8 23 3)	(6 7 5 4)
U2	=	(12 8 4 0)	(13 9 5 1)	(21 23 22 20)
		(12 8 4 0)	(13 9 5 1)	(21 23 22 20)
D2	=	(6 10 14 2)	(7 11 15 3)	(17 19 18 16)
		(6 10 14 2)	(7 11 15 3)	(17 19 18 16)

There is an important optimization which may not be obvious. It turns out we can obtain every state of the cube by combinations which only make use of 6 twists. Why? For example, we can effectuate the L+ twist by performing R- and then rolling the entire cube toward myself. You can think of this as: R+L- simply rotates the entire cube. Similar for the other two pairs of opposite sides. Therefore we can obtain any

permutation of the cube simply using D-, D+, R-, R+, B-, and B+. Or if we want to incorporate the 180° degree twists, we can extend this set of 6 twists to 9 twists D-, D+, D2, R-, R+, R2, B-, B+, and B2. This optimization greatly reduces the amount of memory needed to run such a simulation as a computer program.

We could represent a state of the cube as a 24 character string consisting of the letters `'W'`, `'G'`, `'Y'`, `'B'`, `'O'`, and `'R'`, such as `"BGWRGYYRBYWBGOROWORB"`. However, this would require 24 bytes per state.

Instead, we can represent `"BGWRGYYRBYWBGOROWORB"` as a 24 digit base 6 number, if we map the characters as follows: $W \to 0$, $G \to 1$, $Y \to 2$, $B \to 3$, $O \to 4$, and $R \to 5$.

The maximum such number is

$$\begin{aligned} 555544443333222211110000_6 &= 4,737,649,541,975,930,160_{10} \\ &< 2^{62.04} \\ &< 2^{63} - 1\,. \end{aligned}$$

The cube is encoded as a positive 64 bit integer: 8 bytes.

To rotate apply $D^+ = ((2\,14\,10\,6)(3\,15\,11\,7)(16\,18\,19\,17))$ to cube = `RRRROOOOBBBBYYYYGGGGWWWW`= $555544443333222211110000_6$ = 4737649541975930160_{10}:

$555544443333222211110000_6 \times (2\,14\,10\,6) \to 555544443033232212110100_6$

Computing the digits:

$$\begin{aligned} d_2 &= \left\lfloor \frac{cube}{6^2} \right\rfloor \ \% \ 6 = 0 \\ d_6 &= \left\lfloor \frac{cube}{6^6} \right\rfloor \ \% \ 6 = 1 \\ d_{10} &= \left\lfloor \frac{cube}{6^{10}} \right\rfloor \ \% \ 6 = 2 \\ d_{14} &= \left\lfloor \frac{cube}{6^{14}} \right\rfloor \ \% \ 6 = 3 \end{aligned}$$

$$
\begin{aligned}
cube_{twisted} &= cube + (d_6 \times 6^2 - d_2 \times 6^2) \\
&\quad + (d_2 \times 6^{14} - d_{14} \times 6^{14}) \\
&\quad + (d_{14} \times 6^{10} - d_{10} \times 6^{10}) \\
&\quad + (d_{10} \times 6^6 - d_6 \times 6^6) \\
&= cube + (d_6 - d_2) \times 6^2 \\
&\quad + (d_2 - d_{14}) \times 6^{14} \\
&\quad + (d_{14} - d_{10}) \times 6^{10} \\
&\quad + (d_{10} - d_6) \times 6^6 \\
&= cube + 1 \times 6^2 - 3 \times 6^{14} + 1 \times 6^{10} + 1 \times 6^6 \\
&= 47376410787067206 60_{10}
\end{aligned}
$$

8.7 Homework

There are two homework assignments `homework/bfs.py` and `homework/solverubik.py`; the test cases are found in `tests/bfs_test.py` and `tests/solverubik_test.py`. The student should implement the functions which are incomplete in these files.

Chapter 9

Shortest Paths

Chapter Objectives

- Define weighted graphs and the shortest-path distance function $\delta(u, v)$.
- Apply the relaxation step and justify correctness via the triangle inequality.
- Trace Bellman-Ford, state its $O(|V||E|)$ complexity, and detect negative cycles.
- Trace Dijkstra, state its $O(|E| + |V| \log |V|)$ complexity, and explain why it requires non-negative weights.
- Compare binary-heap and fibonacci-heap for Dijkstra's priority queue.
- Reconstruct a shortest path from source to destination using the $pred$ array.
- Implement single-source shortest path for directed and undirected weighted graphs.

9.1 Single Source Shortest Path on a Weighted Graph

- What is a weighted graph?

- What is a shortest path?
- What are negative weights?
- What are negative cycles?
- What are multiple source shortest paths?

For the moment we are interested in weights we can *add* and compare with $<$.

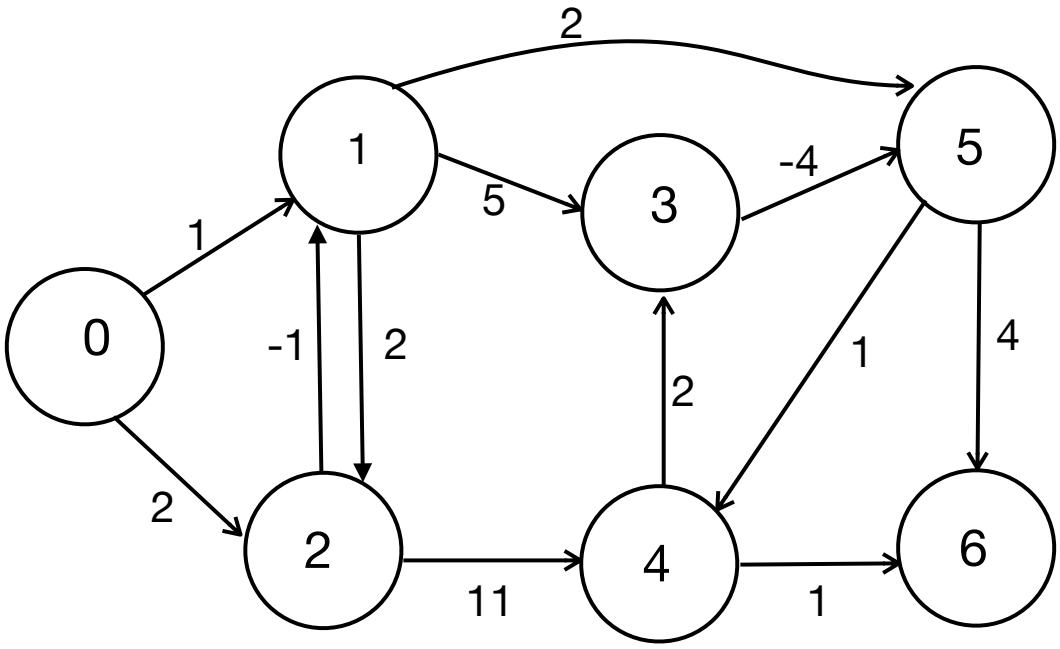
9.2 Weighted graphs

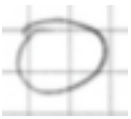
Definition 9.2.1 (*weight*)

If $G = (V, E)$ is a graph, a *weight* is a function $W : E \to S$. Typically S is $\mathbb{N}$ or $\mathbb{R}$.

Let $u \xrightarrow{p} v$ denote the path p from u to v. There may be several paths from u to v, $u \xrightarrow{p_1} v$, $u \xrightarrow{p_2} v$, $\ldots u \xrightarrow{p_m} v$. We can find the *weight of a path* by adding the individual weights of the edges. And since there are only finitely many such paths from u to v, it makes sense to talk about the distance between u and v as being $\delta(u, v) = \min_p w(u \xrightarrow{p} v)$. If there is no path from u to v, we define $\delta(u, v) = \infty$, and $\delta(u, u) = 0$.

Consider the example:


0
1
2
3
4
5
6
1
2
5
-4
-1
2
2
1
4
2
11
1

We write our initial estimate of $\delta(s, v)$ inside  for each $v \in V$. Using the notation,

$$d_s(v) = d(v) = \delta(s, v)\,,$$

we note that the initial estimates are that $d(s) = 0$, and otherwise $d(v) = \infty$.

9.3 Relaxation

We may improve the estimates of A, B, C, and D.

$$\begin{aligned}
d(A) &= \min\{d(A),\ d(s) + w(s, A)\} \\
&= \min\{\infty,\ 0 + 1\} \\
&= 1 \\
d(B) &= \min\{d(B),\ d(s) + w(s, B)\} \\
&= \min\{\infty,\ 0 + 2\} \\
&= 2 \\
d(C) &= \min\{d(C),\ d(A) + w(A, C)\} \\
&= \min\{\infty,\ 1 + 5\} \\
&= 6 \\
d(D) &= \min\{d(D),\ d(B) + w(B, D)\} \\
&= \min\{\infty,\ 2 + 11\} \\
&= 13
\end{aligned}$$

If the current estimate $d(v)$ is known and u is a predecessor of v (i.e., there is an edge (u, v)). Then we can compare $d(v)$ with $d(u) + w(u, v)$, and if

$$d(v) > d(u) + w(u, v)$$

then we can update

$$d(v) \leftarrow d(u) + w(u, v)\,.$$

This relaxation step is possible, because of the triangle inequality, and that δ is a distance function.

$$\delta(v_1, v_2) \leq \delta(v_1, v_3) + \delta(v_3, v_2)$$

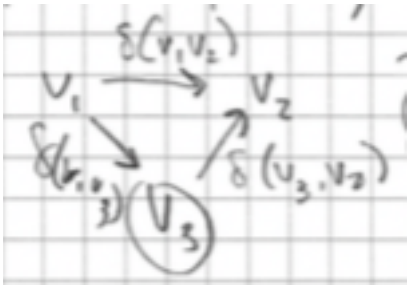
In the example above we calculated $d(D) = 13$, which is effectively the length of the path $S \to B \to D$. But there are several paths from S to D, each with potentially a different length.

$$\begin{aligned} d(S \to B \to D) &= 13 \\ d(S \to B \to A \to E \to D) &= 6 \\ d(S \to B \to A \to C \to E \to D) &= 12 \\ d(S \to B \to A \to B \to D) &= 17 \\ d(S \to B \to A \to B \to A \to B \to A \to B \to D) &= 25 \end{aligned}$$

We may continue, ad hoc updating the estimates. At each iteration we may find another path between two nodes which is shorter than the previous iteration. Since each time we update a value of $d(v)$ the value decreases, and no weights in the graph shown above are negative, the iteration will eventually terminate. The fact that it will terminate with the same values independent of the particular order we relax the vertices is less obvious, but in fact the order does not matter, provided there are no negative cycles.

It may not be clear what the best order to relax the vertices is. In the following sections, we examine various algorithms. They all relax nodes using the same algorithm, the only difference is the order we visit the vertices to relax.

The basic form of the `RELAX` algorithm is the following.



```
1 Function Relax(u,v,w,d,pred)
      Input  : u a vertex
      Input  : v a vertex
      Input  : w weight function
      Input  : d current estimate array
      Input  : pred current predecessor array
2     if d[v] > d[u] + w(u,v) then
3         d[v] ← d[u] + w(u,v)
4         pred[v] ← u
5         Maybe another action depending on caller.
```

Note that the *if* $(d[v] > d[u] + w(u,v))$ *then* may sometimes by written with fewer lines of code.

```
d[v] = min( d[v], d[u] + w(u,v))
```

The problem with this optimization is that you need to update $pred[v]$ in the case that you are changing the value of $d[v]$, and in some cases such as Dijkstra (Section 9.5) you need to also take other action when $d[v]$ changes.

9.4 Bellman Ford

```
1 Function Bellman-Ford(s,V,E,w)
      Input  : s a starting vertex
      Input  : V set of vertices
      Input  : E set of edges
      Input  : w a weight function
2     Initialize
3     repeat |V| − 1 times
4         for (u,v) ∈ E do
5             Relax(u,v,w,d,pred)
      /* Now check for negative weight cycles          */
6     for (u,v) ∈ E do
7         if d[v] > d[u] + w(u,v) then
8             report negative weight cycle
```

```
1 Function Relax(u,v,w,d,pred)
      Input  : u a vertex
      Input  : v a vertex
      Input  : w array of precedents
      Input  : d current estimate array
      Input  : pred current predecessor array
2     if d[v] > d[u] + w(u,v) then
3         d[v] ← d[u] + w(u,v)
4         pred[v] ← u
```

The complexity is $O(|V||E|)$, but since $E = O(|V|^2)$ worst case, Bellman-Ford may have $O(|V|^3)$ complexity. Note the following:

1. If an iteration of the `repeat` loop fails to modify $d[]$ value, then we can abort the iteration early.
2. If the final iteration did not change $d[]$, then the final checking step can be omitted—there are no negative cycles.

9.5 Dijkstra

The Dijkstra algorithm makes use of a heap or priority queue, which has three important operations.

- `INITIALIZE-HEAP` — $O(n)$ — Copy each vertex into the heap, $priority(s) = 0$, otherwise initializing all other priorities to ∞
- `EXTRACT-MIN` — $O(\log(n))$ — remove item with minimum priority
- `REDUCE-MIN` — amortized $O(1)$ — reduce the priority of some item which is not necessarily already the minimum priority.

There are two popular types of heaps: `binary-heap` and `fibonacci-heap`. If you are writing it yourself, `binary-heap` is easier to implement. But if you are using a standard library which provides `fibonacci-heap`, then that's probably a better choice. Why? Because the `fibonacci-heap` offers a `REDUCE-MIN` operation with amortized $O(1)$ time complexity, whereas for the `binary-heap`, `REDUCE-MIN` has $O(\log(n))$ complexity.

```
1  Function Dijkstra(s, V, Adj, w)
       Input  : s a starting vertex
       Input  : V set of vertices
       Input  : Adj adjacency list as mapping V → List[V]
       Input  : w a weight function
2      pred[s] ← None
3      d[s] ← 0
4      for v ∈ V \ {s} do
5          d[v] ← ∞
6      S ← ∅
7      Q ← INITIALIZE-HEAP(V)        // initialize priority
        queue/heap
8      while Q ≠ ∅ do
9          u ← EXTRACT-MIN(Q)          // removing u from Q
10         S ← S ∪ {u}
11         for v ∈ Adj[u] \ S do
12             Relax(u, v, w, d, pred, Q)
```

```
1  Function Relax(u, v, w, d, pred, Q)
       Input  : u, v vertices
       Input  : w array of weights
       Input  : d current estimate array
       Input  : pred current predecessor array
       Input  : Q the priority queue/heap
2      if d[v] > d[u] + w(u, v) then
3          d[v] ← d[u] + w(u, v)
4          REDUCE-MIN(Q, v, d[v])
5          pred[v] ← u
```

The complexity of the Dijkstra algorithm is $O(|E| + |V| \log |V|)$, because

line 7 — is $O(|V|)$.

line 8 — the `while` loops $O(|V|)$ times.

line 9 — is $O(|V| \times \log(|V|))$.

line 11 — loops $O(|E|)$ total, don't be confused about the `while` loop.

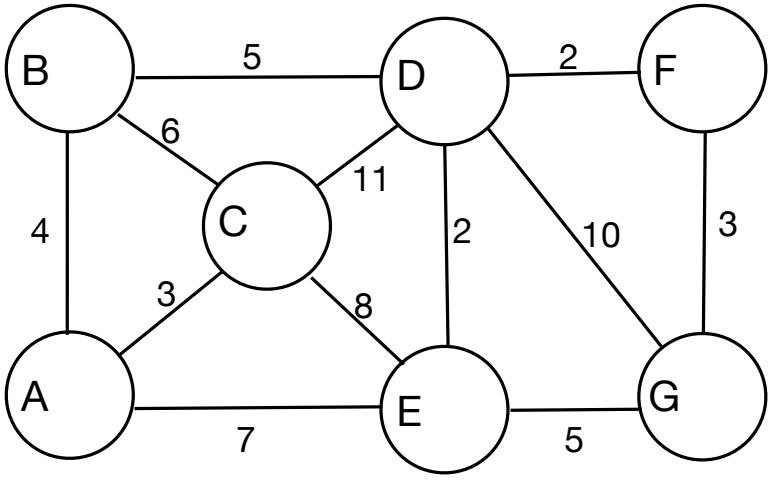
line 12 — is executed $O(|E|)$ with amortized complexity $O(1)$, thus its complexity is $O(|E|)$.

9.6 Dijkstra example

This example is taken from a YouTube video posted by Nataniel Fan, entitled *Dijkstra's Algorithm*.

The Dijkstra algorithm is capable of finding either the shortest path from one vertex to another, or of finding the shortest path from a given vertex to all other vertices. The only difference is do you stop when finding the shortest path to a particular destination, or do you continue until all the vertices are resolved. In this example, we will look at the shortest path between two vertices.

We want to find the shortest path from A to F in the following graph.



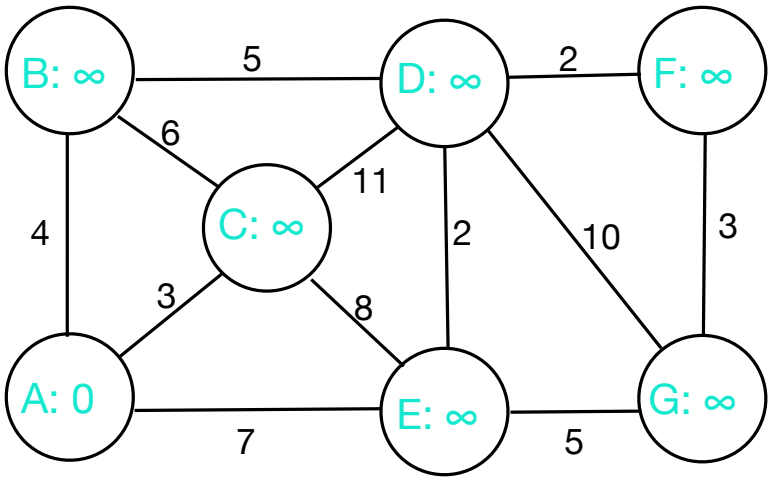
First set $d[A] \leftarrow 0$, because the distance from A to A is 0, and the initial estimate for all the remaining distances is ∞. This operation can be accomplished in different ways.

1. Use a large number for ∞, e.g., `MAX_INT` or 1×10^8.
2. If the programming language can represent ∞, then you can enter all the vertices into the priority queue with $priority = \infty$. In this case $\min\{x, \infty\}$ should return x for any x.

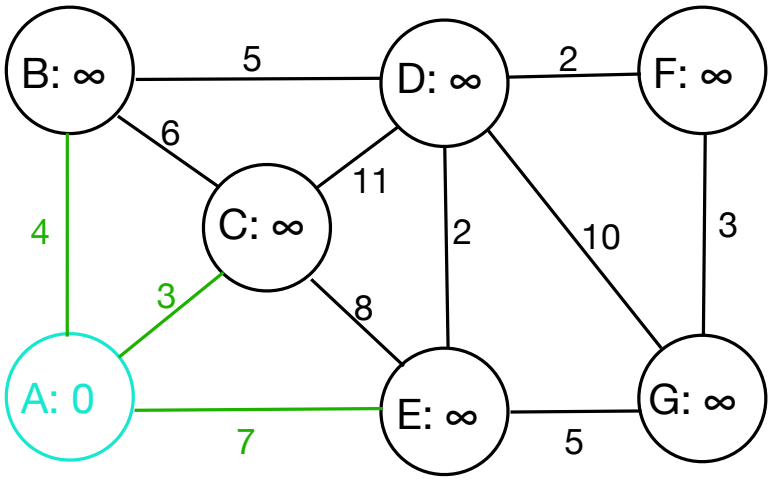
3. If your programming language does not support ∞, then you must always know (or be able to check) whether a given vertex is found in the priority before relaxing it.

So in the latter case, the relax step will contain one case for "not yet registered", and a second case for "already registered".

For this example, let's suppose that the language supports ∞. The graph looks like the following after initialization.



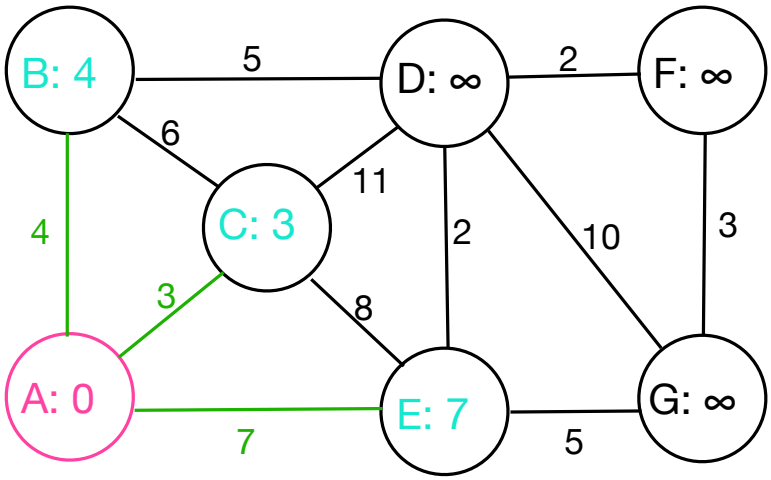
Select the vertex with minimum current $d[]$ value. EXTRACT-MIN $\rightarrow A$, because $d[A] = 0$. We now examine all *outgoing* edges of vertex A, because no vertices have yet been visited. We may now, as always, examine the outgoing edges in any order. The following image shows these outgoing edges with weights 4, 3, and 7.



We relax vertices B, C, and E and mark A as visited. We indicate A as visited as:  .

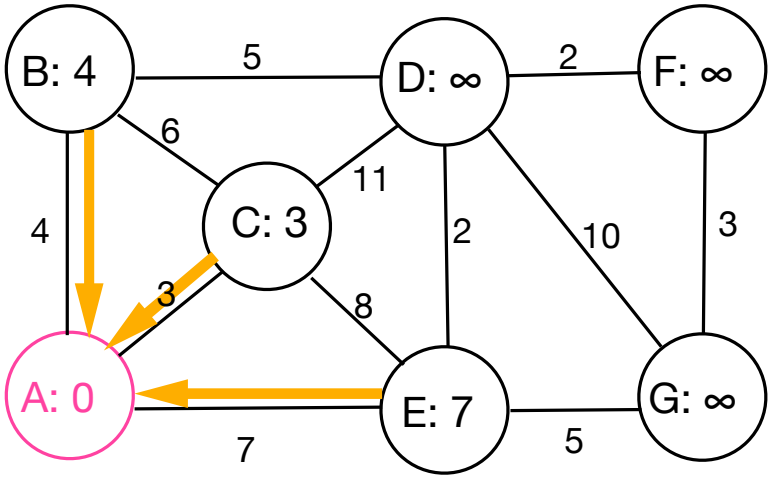
Relaxation involves the following operations:

$$
\begin{aligned}
d(B) &= \min\{d(B), d(A) + w(A, B)\} \\
&= \min\{\infty, 0 + 4\} \\
&= 4 \\
d(C) &= \min\{d(C), d(A) + w(A, C)\} \\
&= \min\{\infty, 0 + 3\} \\
&= 3 \\
d(E) &= \min\{d(E), d(A) + w(A, E)\} \\
&= \min\{\infty, 0 + 7\} \\
&= 7
\end{aligned}
$$



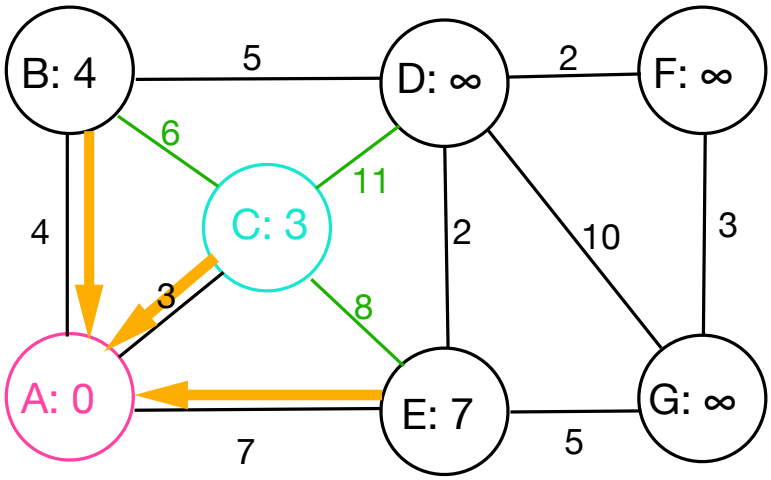
We also update the predecessor every time $d[]$ changes. Since when we examined B, C, and E the values $d[B]$, $d[C]$, and $d[E]$ all changed, we update three predecessors.

$$pred[B] \leftarrow A \qquad B \xrightarrow{pred} A$$
$$pred[C] \leftarrow A \qquad C \xrightarrow{pred} A$$
$$pred[E] \leftarrow A \qquad E \xrightarrow{pred} A$$



Now we again find the minimum value in $d[]$, considering only the

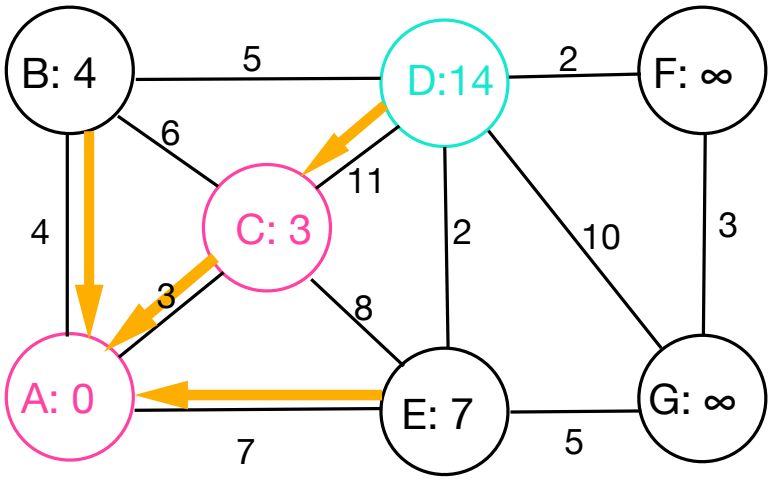
as of yet unvisited vertices. EXTRACT-MIN $\rightarrow C$, because $d[C] = 3$. We now examine all *outgoing* edges of vertex C ignoring edges leading to node A, because A has already been visited. We must examine the edges leading to B, D, and E, with weights 6, 11, and 8, as shown in the image.



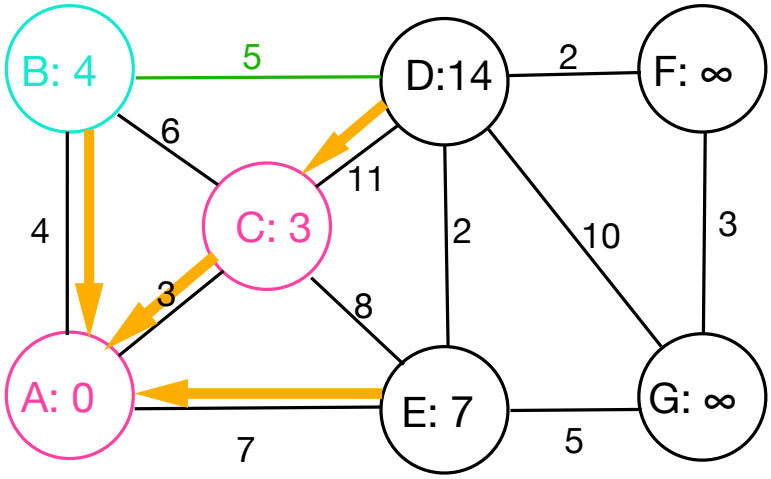
We now relax $d[B]$, $d[D]$, and $d[E]$ as follows:

$$
\begin{aligned}
d[B] &= \min\{d(B), d(C) + w(C, B)\} \\
&= \min\{4, 3 + 6\} \\
&= 4 && d[B] \text{ did not change} \\
d[D] &= \min\{d(D), d(C) + w(C, D)\} \\
&= \min\{\infty, 3 + 11\} \\
&= 14 && d[D] \text{ changed so } pred[D] \leftarrow C \\
d[E] &= \min\{d(E), d(C) + w(C, E)\} \\
&= \min\{7, 3 + 8\} \\
&= 7 && d[E] \text{ did not change}
\end{aligned}
$$

The vertex  goes into the visited list, and the updated graph is as follows:



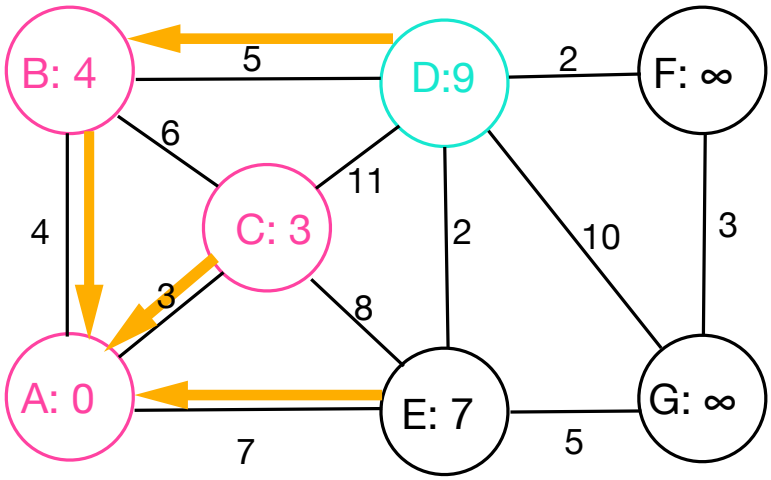
We now find the minimum value ignoring already visited vertices. EXTRACT-MIN $\rightarrow B$, because $d[B] = 4$ is the minimum. We now examine all *outgoing* edges, only one in this case, because A and C have already been visited.



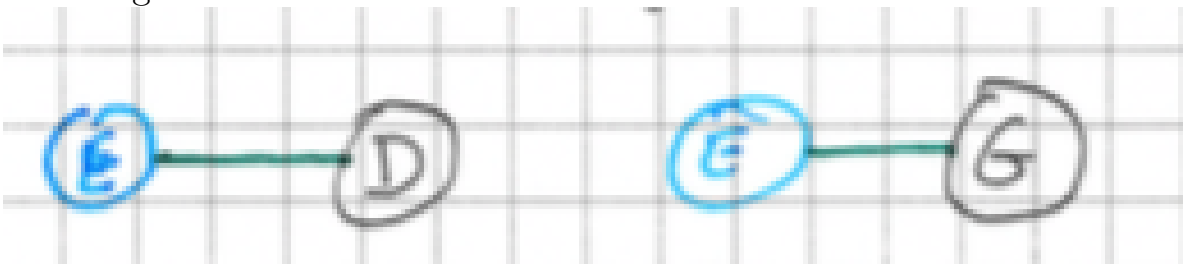
We relax D.

$$\begin{aligned} d[D] &= \min\{d(D), d(B) + w(B, D)\} \\ &= \min\{14, 4 + 5\} \\ &= 9 \qquad\qquad d[D] \text{ changed so } pred[D] \leftarrow B \end{aligned}$$

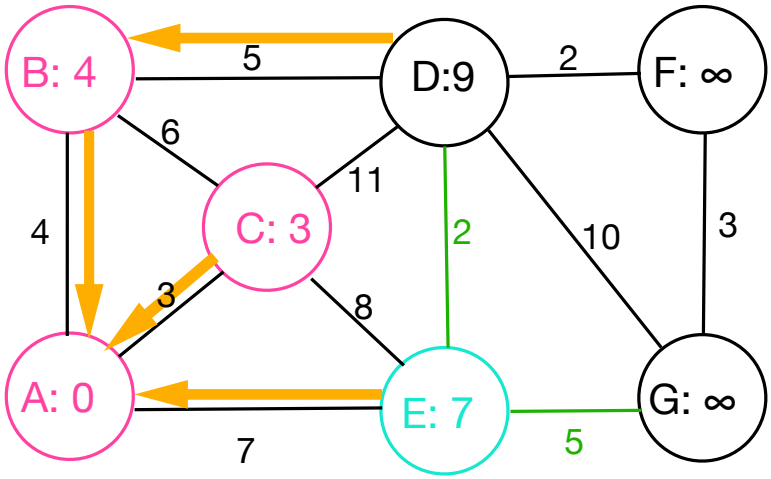
We are finished with B, putting it into the visited list.



EXTRACT-MIN $\rightarrow E$, because $d[E] = 7$ is the minimum. We now visit the edges from E to unvisited vertices.



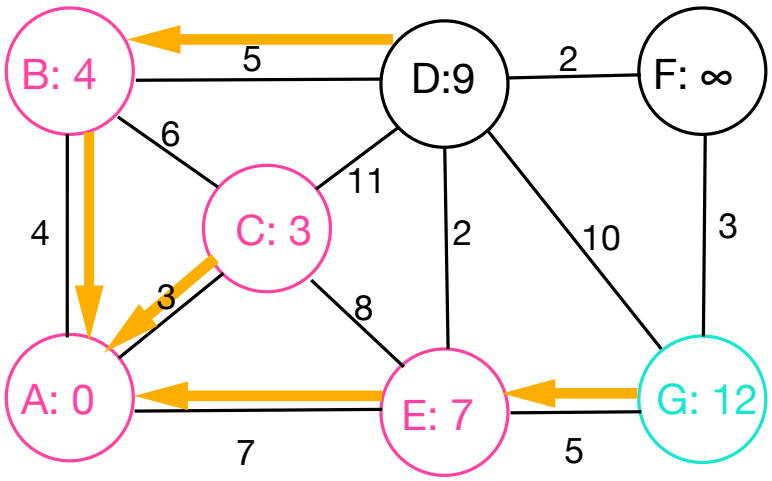
We ignore edges from E to  and  .



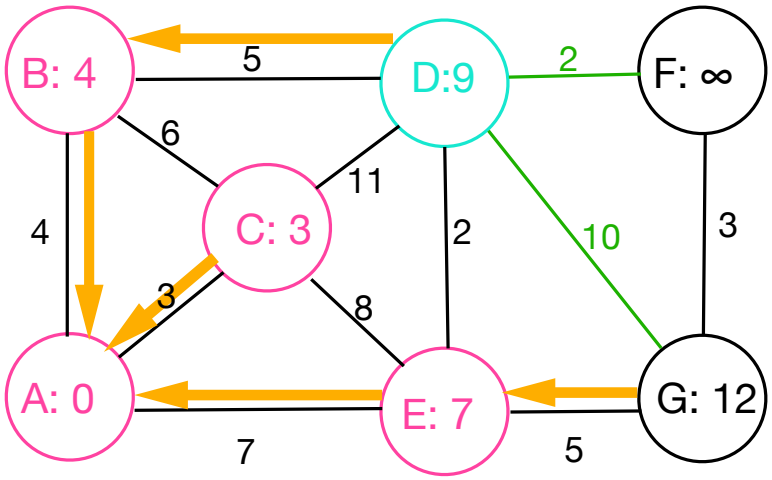
Relaxing D and G:

$$
\begin{aligned}
d[D] &= \min\{d(D), d(E) + w(E, D)\} \\
&= \min\{9, 7 + 2\} \\
&= 9 && d[D] \text{ did not change} \\
d[G] &= \min\{d(G), d(E) + w(E, G)\} \\
&= \min\{\infty, 7 + 5\} \\
&= 12 && d[G] \text{ changed so } pred[G] \leftarrow E
\end{aligned}
$$

We are finished with E, so the visited list is now $\{A, B, C, E\}$.



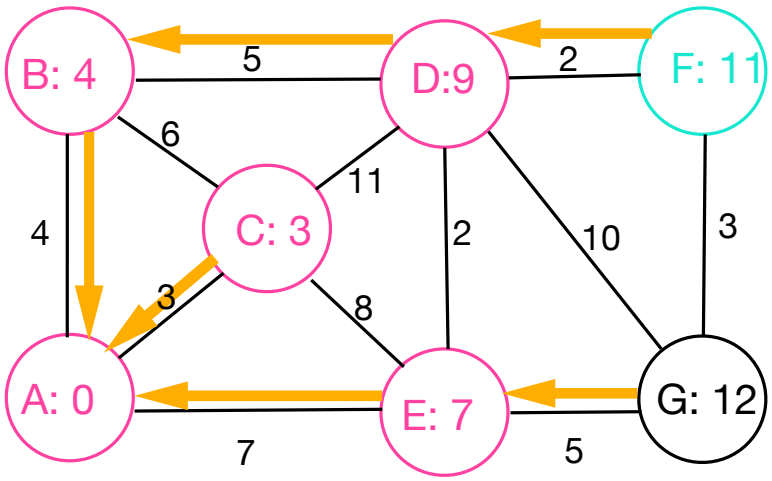
EXTRACT-MIN $\rightarrow D$, because $d[D] = 9$ is the minimum. We now visit the edges from D to unvisited vertices, F, and G. When we relax G nothing changes.



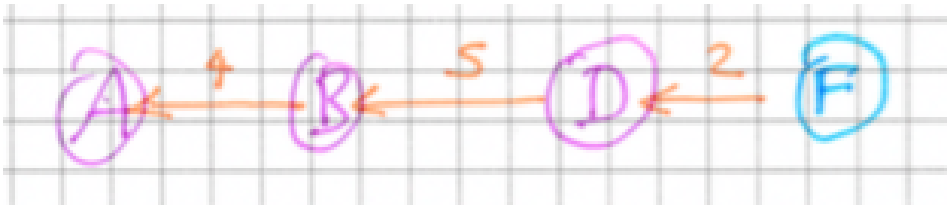
$$
\begin{aligned}
d[F] &= \min\{d(F), d(D) + w(D, F)\} \\
&= \min\{\infty, 9 + 2\} \\
&= 11 \qquad\qquad d[F] \text{ changed so } pred[F] \leftarrow D \\
d[G] &= \min\{d(G), d(D) + w(D, G)\} \\
&= \min\{12, 9 + 10\} \\
&= 12 \qquad\qquad d[G] \text{ did not change}
\end{aligned}
$$

The visited list is updated to also include D.

This time EXTRACT-MIN $\rightarrow F$, because $d[F] = 11$. Since F is the goal vertex, we are done.



The shortest path from A to F has a weight of 11. The actual path may be collected by following the *pred* pointers (in reverse).



9.7 Formulas to remember

$$\sum_{v \in V} degree(v) = \Theta(|E|)$$

$$|E| = O(|V|^2)$$

$$|V| = O(|E|)$$

9.8 Homework

The homework assignment for this chapter is `homework/singlesource.py`; the test cases are found in `tests/singlesource_test.py`. The student should implement the functions which are incomplete in these files.