4.1 Algorithms and Data Structures

4.1.1 Why Data Structures matter (Chapter 1)

Important concepts

Regular array: a list of data elements.

Set: array that doesn't contain duplicates.

Linear search: algorithm where we are going one element at a time from beginning to end to search for an element.

• What are Data Structures and why we should about them

Data Structures are the objects that hold the data. They matter because one particular data structure might suit our application better than another. One way to measure code quality is efficiency, which is measured in the number of steps an algorithm has to take to complete its execution relative to the number of data inputs.

• How to analyze a Data Structure?

There are 4 main operations we can base our analysis on. In the real-world, we'd often look at the most used operations within our application. Those operations are reading, searching, inserting and deleting. For example, let's take arrays and sets: if we check the differences in terms of efficiency for the 4 main operations, we notice that the set's inserting operation takes a total of $2N + 1$ steps in the worst case, while a regular array would only take $N + 1$ steps. That doesn't mean that sets are always worse than arrays, it simply means that they are slower for that particular operation.

4.1.2 Why Algorithms matter (Chapter 2)

Important concepts

Ordered array: array where each element is ordered.

Binary search: efficient algorithm to search in an ordered array, by always removing half of the search space until we find the element.

• **Why are algorithms important?**

Algorithms also need to be considered when building an application. Some operations work better with a particular algorithm and a data structure. Let's say we want to search for a particular number in an array of ordered numbers. We could simply look at each element, one by one, but this doesn't take advantage of the fact the elements are ordered. Instead, we can use a different algorithm called Binary Search, which takes only $O(logN)$ time instead of $O(N)$.

An implementsation of Binary Search

```
my_ordered_arr = [2, 3, 6, 9, 14, 19, 20, 21, 26, 38]

def bs(arr, value):
    midpoint_idx = len(arr) // 2
    steps = 1
    while arr[midpoint_idx] != value:
        if len(arr)==1:
            return 'Item not found in', steps, 'steps'
        if value < arr[midpoint_idx]:
            steps += 1
            arr = arr[:midpoint_idx]
            midpoint_idx = len(arr[:midpoint_idx]) // 2
        else:
            steps += 1
            arr = arr[midpoint_idx:]
            midpoint_idx = len(arr[midpoint_idx:]) // 2
    return 'Item found in', steps, 'steps'
```

4.1.3 The Big O Notation (Chapter 3)

Important concepts

O(1): the algorithm always takes one step no matter how big the data is.

O(N): the algorithm linearly grows as the number of data points increases.

O(log N): the algorithm grows 1 step every time the number of data points is doubled.

Logarithms: they're the inverse of exponents. For example, log2 8 means how much do we have have to keep dividing 8 by 2 until we end up with 1? (=3).

• **What's the Big O notation?**

The Big O notation helps us answer the following question: if there are N data elements, how many steps will the algorithm take? That's not all, though. Knowing how many steps an algorithm is taking is great, but the main motive of Big O, is to **categorize algorithms** by their time complexity so we can **compare them easily**. By default, we compute the Big O by thinking about the worst-case scenario.

4.1.4 Speeding up code using Big O (Chapter 4)

Important concepts

Bubble Sort: It's a sorting algorithm. We compare element at index n with index $n+1$, up until the end of the array. If at any point $n > n+1$, we swap them. Every time we reach the end of the array, we restart this operation but we ignore the last element of the array, since we know this element is in the correct order. We do this until all the array's elements have been exhausted.

O(N^2): rather inneficient algorithm, which often stems from the fact that we use two nested loops. Every time we add an item to the input data, the number of steps = $nb\_of\_items^2$.

• Example situation where Big O can help us create a better algorithm

Let's say we want to create an algorithm that returns *true* if there is a duplicate element in an array of numbers. The most straightforward way to build such an algorithm would be to use two nested loops, where we keep one item in memory in the first loop and compare it with another one in the second loop. While this works, the complexity is $O(N^2)$ in the worst-case. But there's an O(N) solution to that problem. We can loop over the elements in the array once, and place each item in a new array where the item's value is placed at its **index** in the new array. Then, we only have to check if there's something at that index, if there is, it means there's a duplicate.

An efficient duplicate checker

```
def has_duplicate(arr):
    temp = [0] * (max(arr)+1)
    for item_idx in range(len(arr)):
        if temp[arr[item_idx]] == True:
            return True
        else:
            temp[arr[item_idx]] = True
        print(temp)
    return False

list_1 = [1, 2, 5, 2, 8, 9]
print('Has duplicated =', has_duplicate(list_1))

# out:
# [0, True, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
# [0, True, True, 0, 0, 0, 0, 0, 0, 0]
# [0, True, True, 0, 0, True, 0, 0, 0, 0]
# Has duplicated = True
```

• **Bubble Sort, a sorting algorithm explained**

This algorithm performs two kind of operations: comparison and swap. It performs (N-1) + (N-2) + ... + 1 comparisons, and an equal number of swaps (worst-case scenario). Finding the pattern between the number of input items and the steps taken results in numbers that look very close to $O(N^2)$.

Bubble Sort

```
def bubble_sort(arr):
    comparison = 0
    swap = 0
    sorted_ = False
    passthrough = 0
    while sorted_ is not True:
        for idx in range(len(arr)-1-passthrough):
            comparison += 1
            if arr[idx] > arr[idx+1]:
                swap += 1
                arr[idx], arr[idx+1] = arr[idx+1], arr[idx]
        passthrough += 1
        if passthrough == len(arr):
            sorted_ = True
    return arr, swap+comparison
```

4.1.5 Optimizing code with and without Big O (Chapter 5)

Important concepts

Selection Sort: another sorting algorithm. We start by comparing the first element in the array with all the others, and once we've exhausted the list and if we found another number that is smaller, we swap them. Then, restart this operation but starting from the second element, since we know the first is already the smallest from the entire list. We do this once we've checked for every item in the list. True efficiency approaches $O(\frac{N^2}{2})$, but since we're dropping the constants and we only care about the general category of algorithms, we note $O(N^2)$.

• What we care about when computing the Big O

Big O only concerns the general category of algorithms. For ex, Selection Sort is twice as fast as Bubble Sort, but it still has the same Big O notation. Big O describes the rate of growth in terms of steps as the input data increases. For example, if we have an algorithm described in terms of $O(45N)$ and another of $O(5N)$, the constants don't really matter and it's not the question we're trying to answer. We're better off with knowing the general category of this algorithm, $O(N)$, because it is widely different from other categories such as $O(N^2)$.

• Dropping the constant in Big O

Since we're always dropping the constants, it now makes sense how to "calculate" the Big O of an algorithm and figure out what steps matter. If we have an algorithm with a for loop that has a printing line and a comparison line, the "real" time complexity is 1 (loop) + 1 (print) + 1 (compare) = $O(3N)$. But because we're dropping the constant, it's just $O(N)$.

Selection Sort

```
def selection_sort(arr):
    for item_idx in range(len(arr)):
        for next_idx in range(len(arr)):
            if next_idx>item_idx:
                if arr[item_idx] > arr[next_idx]:
                    arr[item_idx], arr[next_idx] = arr[next_idx], arr[item_idx]
    return arr
```

4.1.6 Optimizing for optimistic scenarios (Chapter 6)

Important concepts

Insertion Sort: sorting algorithm that works in $O(N)$ time, hence faster than Bubble Sort and Selection Sort. Works by going through each value, and swapping with the previous one if the previous one is smaller. The current value is inserted where it needs to be, until we go to the next one and we repeat the same process.

Best case & Average case: by default, we consider the worst case scenario to figure out the complexity of an algorithm. However, in the real world, the worst-case scenario is often not the scenario that happens the most often. Hence, we can go a step further and think about the best-case, and average-case scenarios and their own complexity. There's sometimes big differences, as illustrated by Selection sort and Insertion sort. Since Insertion Sort comes with a mechanism to end the pass-through early, its efficiency depends on the form of the input. If the input data is mostly sorted, then Insertion Sort will work better. If the data is ordered randomly, the two algorithms will have similar efficiency.

• **In Big O, we only keep the highest order of N. But why?**
Another major rule of Big O is that we always keep the highest order of N and get rid of the rest. This is because Big O is a tool to categorize algorithms, not a tool that gives us the exact number of steps an algorithm has to take. For example, let's say we feed $N = 100$ data inputs to our algorithm and its « real » time complexity is $N + N^2 + N^3 + N^4$. Computing the number of steps would give us $101,010,100$, which is very similar to $100,000,000$, which itself comes from N^4. So we might as well drop $N + N^2 + N^3$ and only keep N^4.

Insertion Sort

```
def insertion_sort(arr):
  steps = 0
  for item_idx in range(1, len(arr)):
    prev = item_idx-1
    while item_idx>=0 and prev>=0 and arr[item_idx] < arr[prev]:
      steps += 1
      arr[item_idx], arr[prev] = arr[prev], arr[item_idx]
      prev -= 1
      item_idx -= 1
  return arr, steps
```

4.1.7 Big O in everyday code (Chapter 7)

Important concepts

O(2^n): very slow algorithm. Every time we add a new data point, the algorithm doubles in terms of steps.

• Dealing with two data sources in Big O

If we have an algorithm that loops over array M and within this loop, it also loops over over array N, how do we do? It would make sense to describe the notation as $O(N * M)$, however that's not very helpful if we try to compare it to, for example, an O(log N) algorithm. However, we notice that if the size of M is equal to the size of N, then the algorithm is $O(N^2)$. And if M or N's size is equal to 1, then we have an $O(N)$ algorithm. So the time complexity of this algorithm ranges between $O(N)$ and $O(N^2)$.

• An example of an $O(26^n)$ algorithm.

A brute-force generating password algorithm would take 26 steps to generate every possible 1-character password from the alphabet. But if we want a 2-character password, it would take $26 * 26$ steps. If we want 5, it'd take 26^5 steps. So we say this algorithm is of complexity $O(26^n)$.

4.1.8 Hash Tables (Chapter 8)

Important concepts

Hash Tables: they're basically a dictionary, a data structure made for super fast lookup, typically $O(1)$. They take the form of a collection of key/value pairs.

• **A Hash Table use case**

If we had an array of array containing item and price, we'd have to search of the value and then get the price. This is $O(N)$. But hash tables basically take the value, convert it to a memory address by hashing it, and get that value directly. Ideally there's a one-to-one relationship between the value and the hash. If not, the computer will perform linear search to find that value.

• **Typical scenarios to consider for using Hash Tables**

- Searching for an item in a list: we transform the list into a dict, like dict = 'val': true, 'otherval': true and we just have to do dict['val'] to see if the array contains the value or not. We can also get that value directly.

- Organizing conditional logic: we can transform a piece of conditional logic that revolves around paired items into a hash table.

4.1.9 Stack and Queues (Chapter 9)

Important concepts

Abstract Data Types: they are data types that are built upon already-existing native data types. For example, an array is a data type that is commonly built into a programming language (Python, Java, etc..), but we can create a new data type based off that array with a set of restrictions.

Stacks: An example of an LIFO abstract data type. Stacks are arrays with the following restrictions:
1) new item can only be inserted at the end of the array,
2) item can only be deleted at the end of the array,
3) only the last element of the array can be accessed/read.

Queues: Another FIFO abstract data type built on top of arrays. Here are the restrictions imposed on Queues:
1) new item can only be inserted at the end of the array (same as stacks),
2) can only delete the first element of the array
3) only the first element can be accessed/read.

• **LIFO and FIFO explained**

We often talk about operations within abstract data types as being First-In-First-Out (FIFO) or Last-In-First-Out (LIFO). This refers to the behaviour of the data type when data is inserted and removed. FIFO means that the first element to be added to the data type is also the first element to be removed. LIFO means that the last element added to the data type is the first to be removed.

• **Why are Abstract Data Types important**

1. We can easily recognize a stack in others source code and understand what it does.
2. They can prevent potential bugs, since we've heavily constrained the functionality of the original data type used to build our abstract data type.
3. They give us new mental models to tackle new problems. For example, if we were to build a tool that parses programming code to check for missing curly braces, we could build a stack where we check if each character is an opening brace (add it to the stack) or a closing brace (delete it from the stack) and if there's an error, we know the syntax is incorrect.

Stack implementation

```
class Stack:
  def __init__(self):
    self.data = []

  def add(self, val):
    self.data.append(val)

  def remove(self):
    del self.data[-1]

  def read(self):
    return self.data[-1]
```

Queue implementation

```
class Queue:
  def __init__(self):
    self.data = []

  def add(self, val):
    self.data.append(val)

  def remove(self):
    del self.data[0]

  def read(self):
    return self.data[0]
```

4.1.10 Understanding Recursion (Chapter 10)

Important concepts

Recursive function: a function that calls itself.

Base Case: case where the function will *not* recurse. Every recursive function needs a base case, otherwise it'll be stuck in an infinite loop.

• **How to read recursive code?**

1. Identify and understand the base case
2. Identify and understand what happens just before the base case
3. Doing 1) and 2) all the way back to the start of the recursion. If needed, reduce the amount of recursion so that it's small and understandable.

• **How to reduce the amount of recursion to make it easier to understand?**
For example, we can compute the factorial of a number using recursion. Say we want to compute the factorial of 45, this would mean 45 recursions, which is complicated to analyse and understand. So instead, we can figure out the steps taken to compute a much smaller factorial (ex: 3).

• **How do computers "reason" about recursion?** If we call factorial(3), then factorial(3) is executed first. However, the computer doesn't know the value of factorial(3) before it executes factorial(2). And the computer doesn't know the value of factorial(2) before executing factorial(1). So how does it deal with that?

Internally, the computer uses a stack:
1) The computer begins by adding factorial(3) to the stack.
2) Then, it adds factorial(2) to the stack.
3) Finally, factorial(1) is added to the stack. The value of factorial(1) is computed using the base case, and is passed up through the call stack.

• **Recursion in the real world**
Let's say we want to create a function that prints a folder, its subdirectories and its files. We could use a for loop to do this, but we'd need as many for loops as there are subdirectories. Plus, we often don't even know how many subdirectories there are. In this case, recursion really shines: we'd write a program where the base case is "is the current path a file" and if yes, print it, and if not, call the function itself. That would perform a depth-first traversal of all the subdirectories within that folder until all items have been exhausted. Similar example:

An example where recursion shines

```
array_ = [1, [2, 3, [4, 5], 6], 7, [8, 9, 10]]
def print_recursively(my_arr):
    for value in my_arr:
      if isinstance(value, int):
        print(value)
      else:
        print_recursively(value)
```

4.1.11 Writing Recursive code (Chapter 11)

Important concepts

O(N!): extremely inefficient algorithm. For an input size of 4, the algorithm will have to take 4*3*2*1=24 steps. Even less efficient than $O(2^N)$.

- **The multiple categories & tricks of recursive code**

  - **(Category) Repeatedly execute**: here we're just trying to do the same task over and over again in an elegant fashion. For example, listing the files, folders and subfolders within a directory can be solved using recursion. It'd look something like:

```
def print_all_files(path):
  for file_path in path:
    if file_path is a file:
      print(file)
    else:
      print_all_files(file_path)
```

  - **(Trick) Keeping track of an index:** We can use a trick to keep track of an index or something in memory without having to explicitly declare it when calling the function.

```
def double_array(arr, index=0):
  if index>len(arr):
    return
  arr[index] = arr[index]*2
double_array(arr, index+1)
```

  - **(Category) Doing calculations on a problem with similar subproblems**: Say we want to calculate the factorial of a number. We can use a mental model of recursion to easily build a function that solves that problem. If we think about this problem, the factorial of, say, 8, is just $8 * factorial(7)$. Hence, $factorial(7)$ is the sub-problem of $factorial(8)$.

  - **(Category): top-down recursion**: this a new way of thinking about dealing with subproblems. Here's how it works if we want to create a function that computes the sum of each element in the array $[4, 5, 9, 12]$:
    - Identify the subproblem: the array's subproblem is $[5, 9, 12]$, because if we add the sum of those numbers, we'd just have to add the first number, 4.
    - Build the function: using recursion, we can simply do something like:

```
def sum_(arr):
    if len(arr)==1: return arr[0]
    return arr[0] + sum_(arr[1:])
```

• **But why do we need recursion anyways?** The problem listed above only provide elegance, but they can easily be done using the for loops. Here's a problem that can be solved very easily using the new mental model of top-down approach, but would be difficult if we tried with for/while loops:

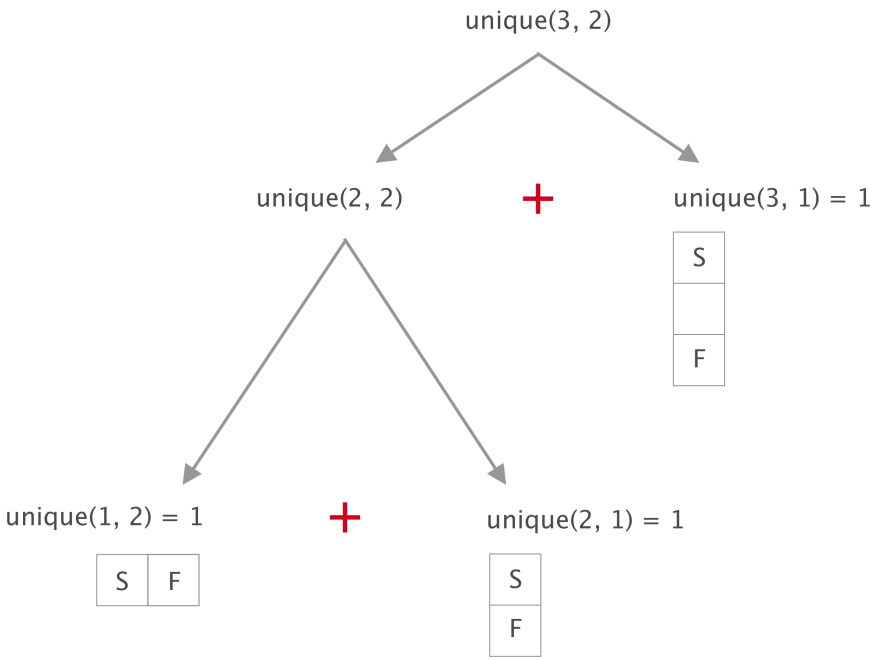
« Given a staircase of N steps and the ability to climb 1, 2 or 3 steps at once, how many possible paths can someone take to reach the top? »

We can solve this problem using the new approach easily: if we have $N = 5$, then the count will be $nb\_of\_steps(5-1) + nb\_of\_steps(5-2) + nb\_of\_steps(5-3)$. The only remaining thing to do, is figure out the count for $N = 3$, $N = 2$ and $N = 1$ but also $N <= 0$. Let's write that in code:

```
def nb_of_steps(N):
    if N <= 0: return 0
    if N == 1: return 1
    if N == 2: return 2
    if N == 3: return 4
    return nb_of_steps(N-1) + nb_of_steps(N-2) + nb_of_steps(N-3)
```

By figuring out the base case FIRST, we can let the computer figure out the rest.

Here's another example, with the "unique paths" problem. Let's say we have a $M * N$ grid and we are trying to figure out **the number of shortest paths that exists to go from the upper left to the bottom right**. Let's think about the base case: it is the case where **either** M or N $= 1$. Then, the number of steps to take is just 1 and it is the shortest path. Let's see this in action:



As we can see, if we start with a simple grid of size $3 * 2$, recursively removing 1 in the column/rows reduces the problem to a straight line (where row/col $= 1$), where the number of shortest paths is always equal to 1.

```
def unique_paths(rows, columns)
  if rows == 1 or columns == 1:
    return 1
  return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
```

4.1.12 Dynamic Programming (Chapter 12)

Important concepts

Dynamic Programming: algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Overlapping subproblems: when a problem is solved by solving smaller versions of that same problem, it's a *subproblem*. But when those subproblems are overlapping, When we need to solve the exact same subproblem(s) more than 1 to solve our final problem, they are *overlapping*. That is rather inneficient as we have to do the same computation, but there is a solution: keeping the outputs in memory and re-using them when we need to.

Let's take a took at a fibonnaci sequence calculator. We could solve this problem using naive recursion, and we would have something like:

An inneficient recursive code

```
def fib(n)
  if n==0:
    return 0
  if n==1:
    return 1
  return fib(n-2) + fib(n-1)
```

This works and is elegant. However, looking closely we notice that to solve, for ex, fib(4), the computer will have to compute $fig(3) + fig(2)$. But to get the answer of $fig(3)$, the computer will have to re-compute fig(2) once more. That's inneficient. If we were to try to compute fig(100), it'd takes hours. This is because this particular has a complexity of $O(2^N)$, one of the worst!

There are two techniques used within DP to help with that issue:

- **Memoization / Top-Down**: we remember what the answer to ALL the computed subproblems are, and store them somewhere. We start from the biggest subprobem (in fig(20) it'd be starting with fig(19) + fig(18)). We might use a hash table for that, because they're $O(1)$ at retrieving information. Example:

An efficient recursive code

```
def fib(n, memo={})
  if n==0:
    return 0
  if n==1:
    return 1
  if n not in memo:
    memo[n] = fib(n-2, memo) + fib(n-1, memo)
  return memo[n]
```

- **Tabulation / Bottom-up**: this just means using another approach than recursion, such as a loop, to solve the problem. We first solve the smallest subproblem of our problem, then go from there to reach the highest subproblem which is often the answer. We still keep the already computed functions in memory to avoid unnecessary computations.

4.1.13 More recursion (Chapter 13)

Important concepts

Partitioning: means taking a value within an array (called the pivot) and making sure that every value that comes before is less than the pivot itself, and values that come after are greater than the pivot itself.

QuickSort: a sorting algorithm. It works using a "divide-and-conquer" approach. It selects a pivot, and recursively divides the two subarrays making sure they are sorted.

A QuickSort implementation

```
def quicksort(arr):
    if len(arr) == 0:
        return []

    less = []
    equal = []
    greater = []

    for elem in arr:
        pivot = arr[-1]
        if elem < pivot:
            less.append(elem)
        if elem == pivot:
            equal.append(elem)
        if elem > pivot:
            greater.append(elem)
    return quicksort(less) + equal + quicksort(greater)
```

4.1.14 Linked Lists (Chapter 14)

Important concepts

Linked List: they are a data structure that work similarly to arrays, except that that the data stored within a linked list doesnt have to take contiguous block of memory. Linked Lists are made of *nodes* that contain the following two information: the data that the node contains as well as a link to the next node.

Doubly Linked List: same as Linked List except that each node has two links instead of one: one for the previous node, and one for the next node. Also, the LinkedList class contains a first_node and a last_node attribute. So accessing/deleting/inserting at the beginning and end of the DLL is always $O(1)$ instead of $O(N)$ like in a standard LL.

• The efficiency of Linked Lists compared to array

- Reading: worst case scenario is if the item we want to read is at the very end of the list, because to read the last element, the computer has to keep track of the current index, and get through the next node one by one. So reading is O(N), which is worse than arrays, O(1).

- Searching: similar to reading, computer has to go through every element until the value of the node we're looking for is found. Worst case is then O(N).

- Inserting: this is where Linked Lists start to become interesting. Inserting in an array means that the computer has to shift all elements that goes after the index. So if the array's length is 10, then inserting takes 10 (shifts) + 1 insert = 11 steps. Inserting at the end of the array however takes only 1 step, since no shifting has to done. With a Linked List, it's the opposite: inserting at index 0 takes 1 step, since all the computer has to do is create a node and point it to the first node. So it's O(1). But inserting at the very end of the Linked List takes N + 1, since the computer first has to find the end of that list and then replace the last link to the new one. So it's O(N+1).

- Deleting: once again, LL are the opposite of arrays for deletion. Deleting at the beginning of the LL just takes one step: take the second node and make it the LL's head. Deleting at the end of the LL however takes N+1 steps, since we first have to find the second-to-last element and then make it the tail of the LL. Deleting at a position that is not head or tail in a LL means that if we have a LL with 4 elements, deleting at index 2 means changing the link from index 1 to point to the node at index 3.

• **Why are Linked Lists useful?**

Let's say we have an algorithm that goes through an array and delete the invalid elements. With an array of size 100, if there's 10 invalid elements, it will take 100 steps (going though every element) + 10*100=1000 (shifting every deleted element) = 1100 total steps. But with a LL, it would only take 100 (going though every element) + 10 (changing the links) = 110 steps.

→ Linked Lists are amazing data structure for INSERTING and DELETING elements as we never have to worry about shifting elements.

Here's a Python implementation of a Doubly Linked List. There is a full code and testing in ./DoublyLinkedList.py.

The Node class

```
class Node:
  def __init__(self, value, next_node=None, previous_node=None):
      self.value = value
      self.next_node = next_node
      self.previous_node = previous_node
```

The DLL class

```
class DoublyLinkedList:
  def __init__(self, first_node: Node, last_node: Node):
      self.first_node = first_node
      self.last_node = last_node
```

Reading methods

```
def read(self, idx):
  current_node = self.first_node
  for i in range(idx+1):
      if i == idx:
          return current_node
      else:
          if current_node.next_node is None:
              return 'Not here, sorry'
          current_node = current_node.next_node
```

Inserting methods

```
def insert_at_start(self, val):
  new_node = Node(value=val, next_node=self.first_node)
  self.first_node.previous_node = new_node
  self.first_node = new_node

def insert_at_end(self, val):
  new_node = Node(value=val, previous_node=self.last_node)
  self.last_node.next_node = new_node
  self.last_node = new_node

def insert_at_any(self, idx, val):
  # check that we're not dealing with the first OR last node
  if self.read(idx) == self.last_node:
      self.insert_at_end(val)
  elif self.read(idx) == self.first_node:
      self.insert_at_start(val)
  else:
      # otherwise, insert:
      _new_node = Node(value=val)
      _prev_node = self.read(idx-1)
      _next_node = self.read(idx)
      _prev_node.next_node = _new_node
      _new_node.next_node = _next_node
      _next_node.previous_node = _new_node
      _new_node.previous_node = _prev_node
```

Deleting methods

```
def delete_last(self):
  second_to_last = self.last_node.previous_node
  second_to_last.next_node = None
  self.last_node = second_to_last
```

Searching methods

```
def search(self, val):
  current_node = self.first_node
  index = 0
```

```
    while current_node.value != val:
        index += 1
        if current_node.next_node is None:
            return 'Not here, sorry'
        current_node = current_node.next_node
    return index
```

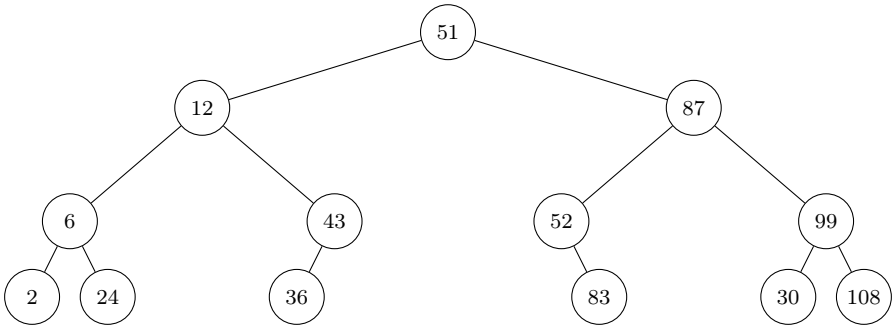
4.1.15 Binary Search Trees (Chapter 15)

Important concepts

Tree: a Tree is a node-based data structure where each node can have multiple links to other nodes.

Binary Search Tree: an abstract data type that maintains order, has fast search, deletion and insertion. The « binary » stems from the fact that each node has either 0, 1 or 2 children. The « search » adds 2 additional constraints: each node has at most one left child (that is less than the parent node) and one right child (than is greater than the parent node.). Just like Linked Lists, BST are node-based data types, hence we can build them by creating a TreeNode class.

Balanced tree: A tree is « balanced » when the amount of leaf nodes on the right hand side of the tree is the samne as the amount of left hand side leaf nodes. Having a perfectly balanced BST allows to have to speedups we're looking for. In a totally unbalanced tree, such as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, the effiency of searching is O(N) and it's the worst-case scenario.



• **Analyzing the 3 main operations on the BST above**

- Searching: searching is extremely efficient in a BST. Here is how it works: say we're looking for the value "52": we start by evaluating the root node, "51". If "52" is bigger than "51", go the right side. Now we're at "87", check if "52" is $>$ or $<$ than "87". It's less, so go the left. We found our value. Time complexity is $O(logN)$, since we're ignoring half the remaining leafs after each evaluation.

- Inserting: inserting involves searching for the best node to attach the value to. Say we want to insert 17 in the above tree. We'd search for 17 and end up at the bottom node "36". We'd simply add a left node to 36. Inserting is then $O(logN + 1)$ (searching + 1 insertion).

- Deleting: deleting is the hardest of those operations. Let's break down how deletion works given the situation:

• If the node has 0 children (ex: 36), find it and delete it.
• If the node has 1 child (ex: 43), replace the child node with that node.
• If the node has 2 children (ex: 87), find the successor node: to find it we select the right hand side of that node and continually moves leftward. So the successor of 87 is 30. Replace 87 with 30, and of course delete the 30 at the bottom of the tree.
• If the node has 2 children (ex: 51) and the successor node is a right child (52 - 83), then take the successor node (52), replace 51 with it, then replace 52 with 83 to not have a hanging child.

• Traversing a BST

With In-Order Traversal, a technique to traverse the tree. It can, for example, print things in order since the structure of a BST keeps order. It's done recursively, and here are the steps:

1. Start at the root node, and call itself on the node's left child until we hit a node that doesn't have any left child.
2. Visit the node (for example, print its value).
3. Call itself on the node's right child, until we hit a node that doesn't have a right child.

• How are BST useful?

Let's say we want to store and manipulate ordered data often. We have an app that should be able to quickly print out a list in order, allow for constant changes in the list and allow the user to search for a title within the list. BST are the perfect tool for this use-case.

• BST implementation

There is the full code at ./BinarySearchTree.py with tests.

Node class

```
class TreeNode:
    def __init__(self, value, leftChild=None, rightChild=None):
        self.value = value
        self.leftChild = leftChild
        self.rightChild = rightChild

    def __repr__(self):
        return str(self.value)
```

BST class

```
class BST:
    def __init__(self, root_node):
        self.root_node = root_node
```

Methods to find the parent node and the successor node

```
def find_parent_node__(self, value, node):
  if (node.leftChild and node.leftChild.value == value) or
  (node.rightChild and node.rightChild.value == value):
    return node
  else:
    if node.value > value:
      return self.find_parent_node__(value, node=node.leftChild)
    elif node.value < value:
      return self.find_parent_node__(value, node=node.rightChild)

def find_successor_node__(self, node):
  if node.leftChild is None:
    return node
  return self.find_successor_node__(node.leftChild)
```

Searching methods

```
def search(self, value, node):
  # 2 base case:
  if node is None:
    return node
  if node.value == value:
    return node

  # recurse:
  if value < node.value:
    return self.search(value, node.leftChild)
  elif value > node.value:
    return self.search(value, node.rightChild)
```

Inserting methods

```
def insert(self, value, node):
  if value < node.value:
    if node.leftChild is None:
      node.leftChild = TreeNode(value)
      return
    self.insert(value, node.leftChild)
```

```
    else:
      if node.rightChild is None:
        node.rightChild = TreeNode(value)
        return
      self.insert(value, node.rightChild)
```

Deleting methods

```
def delete(self, value):
  node = self.search(value, node=self.root_node)
  parent = self.find_parent_node__(node.value, node=self.root_node)

  # ------------ if node has 0 child, delete it
  if node.leftChild is None and node.rightChild is None:
      if node.value < parent.value:
          parent.leftChild = None
      else:
          parent.rightChild = None
      return

  # ------------ if node has 1 child only
  # - case where it's only 1 left child
  elif node.leftChild is not None and node.rightChild is None:
      parent.rightChild = node.leftChild
      return
  # - case where it's only 1 right child
  elif node.leftChild is None and node.rightChild is not None:
      parent.leftChild = node.rightChild
      return

  # ------------ if node has 2 children
  # step 1: find successor node
  move_one_right = node.rightChild
  successor_node = self.find_successor_node__(move_one_right)

  # step 2: delete that successor node:
  parent_successor = self.find_parent_node__(
    successor_node.value, node=self.root_node)
  if successor_node.value < parent_successor.value:
```

```
        # remove left child
        parent_successor.leftChild = None
    else:
        # remove right child
        parent_successor.rightChild = None

    # step 3: replace node with successor node
    node.value = successor_node.value

    # step 4: (optional), if the
    # successor node has a right child
    if successor_node.rightChild:
        parent_successor.leftChild = successor_node.rightChild
```

Traversing methods

```
def traverse(self, node):
    if node is None:
      return 'end'
    self.traverse(node.leftChild)
    print(node.value)
    self.traverse(node.rightChild)
```

4.1.16 Heap (Chapter 16)

Important concepts

Priority Queue: queue with the following constraints:

1. Only delete and access data from the beginning of the queue.
2. When inserting data, the queue remains sorted.

Binary Tree: special kind of tree where each node has a maximum of two child nodes.

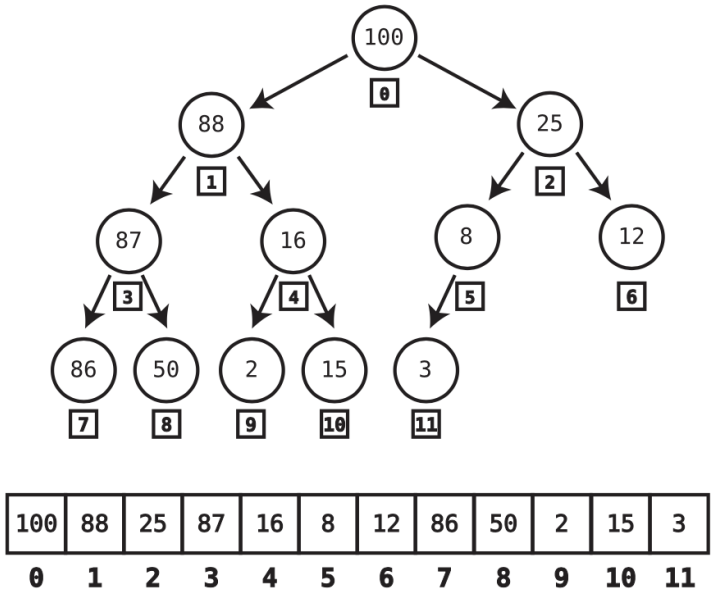
Heap: perfect tool for implementing Priority Queues. It is a binary tree that has those constraints:

1. each child node is inferior than its parent.
2. the tree is complete: each node is present in the tree (there's no empty nodes). This is different from having a balanced tree.

• Insertion and Deletion in a Heap

Let's take a look at insertion and deletion in a Heap. Because we'll need to figure out an algorithm for finding the right-most bottom row spot in the tree, let's see how this is done first and then move on to the 2 operations.

Finding Bottom Right Element Technique: Heaps are usually implemented using arrays, since it's easy (and fast) to find the the last element of that array. Example:



Here can easily see that the last element of that tree is at index 11, i.e. 3. How to figure

out who is children/parent of who? It's easy using those two formulas:
- To find the left child of node: $(index * 2) + 1$
- To find the right child of node: $(index * 2) + 2$
- To find the node's parent: $\frac{(index-1)}{2}$

Example: let's take 16. Its left child node is $(4 * 2) + 1 = 9$. Its right side node is $(4 * 2) + 2 = 10$.

Insertion steps:

Let's say we want to add the number 40.

Step 1: find the right-most empty spot in the bottom row of the tree using the technique explained above. In this case it'd be at index 12. Insert it here for now.

Step 2: compare this node with its parent node. If superior, swap, and inferior, stop. Ex: $40 > 8$, so we swap them. Then, $40 > 25$ so we swap them too. Finally, $40 < 100$ so we stop. The value is added.

Deletion steps:

Note: remember that we only ever delete the root node in a Heap.

Step 1: swap right bottom node with the root node. So we swap 3 with 100.

Step 2: we trickle down the 3 where it's most appropriate: if left or right value is higher, we swap. Until the 3 is higher than both of its children. So we'd swap 3 and 88. Then 3 and 87. Then 3 and 86 and we're done.

- **Implementing a Heap**

The Heap class

```
class Heap:
  def __init__(self):
      self.tree = []
```

Methods to find the parent, left child, right child and biggest child

```
def __find_parent_node_index(self, idx):
  return (idx-1) // 2

def __find_left_child(self, idx):
  return (idx*2) + 1

def __find_right_child(self, idx):
  return (idx*2) + 2

def __find_biggest_child_idx(self, idx):
  left_child_idx = self.__find_left_child(idx)
```

```
    right_child_idx = self.__find_right_child(idx)
    if self.tree[left_child_idx] > self.tree[right_child_idx]:
      return left_child_idx
    else:
      return right_child_idx
```

Methods for inserting

```
def insert_(self, value):
    self.tree.append(value)
    last_idx = len(self.tree)-1
    parent_idx = self.__find_parent_node_index(last_idx)
    while value > self.tree[parent_idx]:
      self.tree[last_idx], self.tree[parent_idx] =
        self.tree[parent_idx], self.tree[last_idx]
      last_idx = parent_idx
      parent_idx = self.__find_parent_node_index(last_idx)
      if parent_idx < 0:
        break
```

Methods for deleting

```
def delete_(self):
    # swap
    self.tree[0] = self.tree[-1]
    del self.tree[-1]

    # trickle down
    root_node_idx = 0
    highest_child_idx = self.__find_biggest_child_idx(root_node_idx)
    while self.tree[root_node_idx] < self.tree[highest_child_idx]:
      # swap
      self.tree[root_node_idx], self.tree[highest_child_idx] =
      self.tree[highest_child_idx], self.tree[root_node_idx]
      root_node_idx = highest_child_idx
      # find next children.
      try:
        highest_child_idx = self.__find_biggest_child_idx(root_node_idx)
      except IndexError:
        return
```

4.1.17 Trie (Chapter 17)

Important concepts

Trie: tree-based abstract data type that is ideal for building features such as a phone autocomplete. It is a collection of nodes that points to other nodes. We can implement a trie using hash tables, where each node contains other nodes of the trie. Tries are very efficient when they're using hash tables, since accessing is always $O(1)$.

- **Implementing a Trie (full code can be found at ./Trie.py)**

A trie class

```
class Trie:
  def __init__(self):
    self.root = {}
```

Methods for inserting

```
def insert(self, value):
  current_node = self.root
  for char in value:
    if char in current_node:
      current_node = current_node[char]
    else:
      current_node[char] = {}
      current_node = current_node[char]
  current_node['*'] = None
```

Methods for inserting

```
def search(self, value):
  current_node = self.root
  for char in value:
    if char in current_node:
      current_node = current_node[char]
    else:
      return 'not found'
  return current_node
```

Methods for autocomplete

```
def collectWords(self, root, words=[], current_word=''):
  for k, child in root.items():
    if k == '*':
      words.append(current_word)
    else:
      self.collectWords(
        child,
        words=words,
        current_word=current_word+k)
  return words

def autocomplete(self, value):
  root = self.search(value)
  if root is None:
    return 'not found'
  return self.collectWords(root, words=[], current_word=value)
```

4.1.18 Graphs (Chapter 18)

Graph: data structure that specializes in relationships. It can have cycles (nodes linking to each other in an endless loop), and every node can or cannot be connected to each other. In a graph, a node is called a **Vertex** (plural: **Vertices**). And the link between each node is a called an **Edge**. Vertices that have an edge between them are called **Adjacent**. A graph is said to be **Directed** if there's a direction to the edges between two vertices. A graph is **Connected** if there's a possibility of traversing the entire graph (all the vertices) using the edges. A **weighted graph** has a weight to each edge.

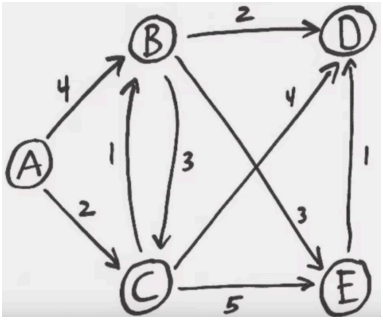
Search in graphs, to...
• Find a particular vertex within the graph
• Discover whether two vertices are connected
• Perform an operation on all the vertices within the graph. This works only if the graph is connected.

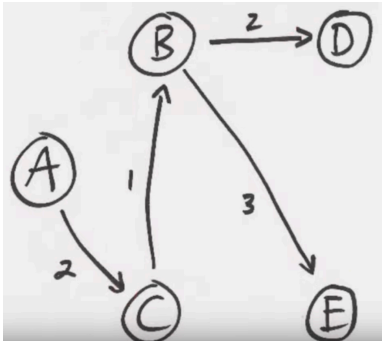
• How to search a graph
- Depth-First Search: search technique for graphs. DFS visits a node's first adjacent vertex, and then this adjacent vertex's adjacent vertex, until it reaches a node that has all the vertex visited. Efficiency is $O(V + E)$.
- Breath-First Search: another search technique for graphs. BFS visits the starting vertex and all of its adjacent vertices first, and then moves on the one of those adjacent vertex and visit its immediate adjacent vertices first too, until all the vertices have been visited. Usually implemented using a Queue abstract data type. Efficiency is $O(V + E)$.

• DFS vs. BFS
BFS traverses all the vertices closest to the starting one, before moving further. DFS however traverses and moves as far away as it can from the starting vertex. Choosing the right traversing algorithm depends on how we answer the following question: *do we want to stay close to the starting vertex during the search, or do we want to move as far away as we can?*

• **Dijkstra's Algorithm**

Works on weighted directed graphs and helps us find the shortest path from **one** node (that we choose) to **all** the other nodes. We'd typically have a graph such as:





Source: Michael Sambol (2014) *Dijkstra's algorithm in 3 minutes — Review and example.* [online video]
Available at: www.youtube.com/watch?v=_lHSawdgXpI.

Let's see what the steps are for this graph:

1. 1. We pick a starting node we're interested in. Let's select A. Let's create the table:
   A: 0
   B: ∞
   C: ∞
   D: ∞
   E: ∞
   Already-calculated-nodes: [A]
   with every value and its corresponding shortest path "cost".

2. We update the table with all the adjacent vertices. So we have:
   A: 0
   B: $4 < \infty \rightarrow 4$
   C: $2 < \infty \rightarrow 2$
   D: ∞
   E: ∞

3. We select the smallest cost from the two adjacent vertices, which is C. From C, we examine all its adjacent vertices and update the table, remembering to always 2 since the distance between A and C is 2:
   A: 0
   B: $1+2=3 < 4 \rightarrow 3$
   C: 2
   D: $4+2=6 < \infty \rightarrow 6$

E: $5+2=7 < \infty \rightarrow 7$

Already-calculated-nodes: [A, C]

4. We choose the smallest path from the ones we just visited: E > D > B. From B, we repeat this process, keeping track of the distance between A and B. It is now 3. Let's see what we have now:

   A: 0

   B: 3

   C: $3+3=6 > 2 \rightarrow 2$

   D: $3+2=5 < 6 \rightarrow 5$

   E: $3+3=6 = 6 \rightarrow 6$

   Already-calculated-nodes: [A, C, B]

5. We do the same thing, selecting the smallest path from the unvisited nodes. It is D. There's no edge from D, so we don't anything.

   Already-calculated-nodes: [A, C, B, D]

6. Finally, we select the last vertex, E, and do the same process:

   A: 0

   B: 3

   C: 2

   D: $6+1=7 > 5 \rightarrow 5$

   E: 6

   Already-calculated-nodes: [A, C, B, D, E]. We've now seen all the vertices and the algorithm stops.

4.1.19 Space Complexity (Chapter 19)

Space Complexity: similar to Time Complexity, space complexity deals with the following question: *if there are N data elements, how many units of memory will the algorithm consume?.*

• Why care about Time Complexity?

We often have to make trade-offs, and knowing the Time and Space complexity of an algorithm helps us make the best possible decision given our situation. To calculate the Space Complexity, we simply need to figure out how many units of memory the algorithm takes to solve the problem. For example, if an algorithm takes an array of size N and creates two sub-arrays of size N/2 + N/2, the algorithm takes N additional space to execute. Note that algorithms doing in-place modifications never take extra memory, so their Space Complexity is O(1).

• Recursion and Space Complexity

In a recursive function, every recursive call adds 1 unit of memory to the calculations of the Space Complexity.

4.1.20 Techniques for code optimization (Chapter 20)

Here, we present a few useful techniques that can help optimize a slow algorithm by making its time complexity better.

• Steps to optimize an algorithm

We can follow those steps whenever we've built an algorithm and we sense that it could be further optimized:

1. Look at current implementation and find the Big O.
2. Determine best possible Big O. For example, if we know we have to do something for every element within an array, the best possible Big O is $O(N)$.
3. Change algorithm or data structure to reach the goal set at step 2.

• Using a greedy algorithm

Greedy algorithms are a class of algorithms that aim to solve a particular problem by computing a local solution to a problem. A local solution is a solution that appears to be the best at a particular point in time, and we hope that by iteratively computing that solution, we might arrive at a global solution.

For example, let's say we want to create an algorithm that find the highest element in an array. An intuitive approach would look like this:

An $O(N^2)$ solution to the problem

```
def findMax(arr):
  highest = 0
  for a in range(len(arr)):
      for b in range(len(arr)):
          if arr[b] > arr[a]:
              highest = arr[b]
  return highest
```

While the above solution works, it's extremely inneficient. The greedy approaches things differently: what if we said that, at first, the highest element is 0. Then, we compare that with the 1st element in the array, and check if it's higher. If it is, we replace it with our *highest* variable, and carry onto the 2nd element. We do this until the end of the array, and return the value of *highest*.

An $O(N)$ solution to the problem

```
def findMax(arr):
  highest = 0
  for n in arr:
    if n > highest:
      highest = n
  return highest
```

Here we're simply comparing our current, greedy and local *highest* with the current element of the array. If we find a better solution so far, we replace it. This is much more efficient.

• **Finding patterns**

We can find interesting patterns - which in turn might help make our algorithm simpler and faster -, by creating a 2-column table of inputs and their respective outputs and then taking a closer look at the whole. We might perhaps find an interesting pattern, which would make building or optimizing the algorithm easier.