Robert Morgan
Dec. 2020, Jan. 2021

Programming Interview Practice

This document contains problems, solutions, and reflections on concepts typically tested in coding interviews. The problems are done without the aid of an IDE or text editor to simulate the “whiteboard style” questions of a coding interview.

Repository of Materials: GitHub Repo

Language of Choice: Python

Table of Contents

Robert Morgan
Dec. 2020, Jan. 2021

Programming Interviews Exposed: Secrets to Landing Your Next Job

Book by John Mongan and Noah Suojanen

Chapter 3: Linked Lists

Problem: *Stack Implementation*. Implement a stack using either a linked list or dynamic array (justify your decision). Design the interface to your stack to be complete, consistent, and easy to use

Solution

I choose to implement the stack as a linked list. The disadvantage of a linked list is that you need to iterate through the list to find an element, however, since we are implementing a stack, we only need to access the first element. Dynamic arrays have convenient, index-based element access for interacting with the top of the stack, in practice will be less efficient for a stack. The reasoning here is that arrays store their data in contiguous memory locations, so every time elements are popped or pushed onto the stack, the memory addresses of all elements will need to be updated, making these operations O(n). A linked list will only require the adjustment of a couple pointers, which will be O(1) complexity.

```
class DoublyLinkedListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class Stack:
    def __init__(self):
        self.head = None

    def push(self, data):
        new_node = DoublyLinkedListNode(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def pop(self):
        if head is None:
            raise IndexError(“Cannot pop from an empty Stack”)
        data_to_pop = self.head.data
        self.head = self.head.next
```

```
        self.head.prev = None
        return data_to_pop

    def peek:
        if head is None:
            raise IndexError(“Cannot peek from an empty Stack”)
        return self.head.data
```

Notes based on book solution:

- A dynamic array doesn’t require additional memory allocation for pointers, so if the elements are all small integers (require much less memory than a pointer) then a dynamic array may use less memory space
- LinkedLists are also much less complicated than dealing with dynamic resizing, so that is another advantage, especially in an interview

Problem: *Maintain Linked List Tail Pointer*. `head` and `tail` are global pointers to the first and last elements, respectively, of a singly linked list of integers. Implement functions to `delete` and `insert_after`. The argument to `delete` is the element to be deleted. The two arguments to `insert_after` give the data for the new element and the element after which the new element is to be inserted. It should be possible to insert at the beginning of the list by calling `insert_after` with `None` as the element argument. These functions should return 1 if successful and 0 if unsuccessful.

Solution

Not really much to talk about, let’s just code.

```
class Element:
    def __init__(self, data):
        self.data = data
        self.next = None

def locate(element):
    current, prev = head, None
    while current.data != element.data:
        if current.next is None:  # Assumes tail.next is None
            raise ValueError(f”{element} not in linked list”)
        prev = current
        current = current.next
    return prev, current


def delete(element):
    if element is head:
        head = head.next
        return 1

    try:
```

```
        current, prev = locate(element)
    except ValueError:
        return 0  # Unsuccessful

    if current is tail:
        tail = prev

    prev.next = current.next
    del current
    return 1

def insert_after(element, data):
    new_element = Element(data)

    if element is None:  # Insert at beginning
        new_element.next = head
        head = new_element
        return 0
    elif element is tail:  # Insert at end
        current = tail
        tail = new_element
    else:  # Lookup element
        try:
            current, _ = locate(element)
        except ValueError:
            return 0  # Unsuccessful

    new_element.next = current.next
    current.next = new_element
    return 1
```

Notes based on book solution:

- Make sure to update head and tail global pointers if they are affected by the insertion or deletion
- Account for corner cases of the list being empty or exceeding some maximum length that will overflow memory

Problem: *Bugs in `remove_head`.* Find and fix the bugs in the following function that is supposed to remove the head element from a singly linked list:

```
def remove_head(head):
    del head
    head = head.next
```

Solution

The bug in this code is that `head` is deleted prior to the pointer to the next node being utilized to set the new head of the linked list. Therefore, the interpreter will raise a `NameError` on the variable "`head`" when the second line of the function is executed. To fix the bug, we need to

remove the `del head` line and leave the removal of the old head from memory up to garbage collection, or utilize a temporary variable to point to head such that we can remove it from memory ourselves. If the individual elements in this list take up large chunks of memory, the second approach is preferred such that we can ensure the freed-up memory is available immediately.

```
def remove_head(head):
    head = head.next

# OR to ensure memory is immediately free

def remove_head(head):
    temp = head.next
    del head
    head = temp
```

Notes based on book solution:

- Probably good practice to check that the object passed to the function has the next attribute and really is a pointer to the head of a linked list

Problem: *Mth-to-Last Element of a Linked List*. Given a singly linked list, devise a time- and space-efficient algorithm to find the mth-to-last element of the list. Implement your algorithm, taking care to handle relevant error conditions. Define mth-to-last such that when m=0, the last element in the list is returned.

Solution

The brute force approach is to iterate through the list twice, the first time counting the elements and the second time accessing the mth-to-last element. This is space-efficient but not quite time-efficient (we can probably get creative and do better than O(2n)).

A slightly better solution would be to iterate through the list fully once, and load each element into a hashtable with the index as the key. This solution would only require one iteration through the list and one constant time lookup using the hashtable, making it O(n+1), but now we run into the problem of being space-inefficient since the hashtable will require just as much memory as the original list. Plus inserting each element into the hashtable is n O(1) operations, so this is basically O(2n+1) even with the hashtable.

My proposal is to use two pointers simultaneously. Start the first pointer at head. If the end of the list is reached before we have gotten to the mth element, handle the error case that the list doesn't have an mth-to-last element. If the list has enough elements, start the second pointer at head once the first pointer reaches the mth element. Then step through the list incrementing both pointers at the same time. Once the first pointer reaches the tail, return the element where the second pointer is pointing.

Robert Morgan
Dec. 2020, Jan. 2021

```
def get_mth_to_last(head, m):
    if m < 0:
        raise ValueError("Value of m must be nonnegative")

    lead_pointer = head
    if lead_pointer is None:
        raise ValueError("Linked List cannot be empty")

    counter = -1
    rear_pointer = None
    while lead_pointer.next != None:  # Indicating we're at the end
        if rear_pointer is None:
            counter += 1
            if counter == m:
                rear_pointer = head
        lead_pointer = lead_pointer.next
        rear_pointer = rear_pointer.next

    if rear_pointer is None:  # Indicating list was shorter than m
        raise IndexError("linked list must contain at least m elements")

    return rear_pointer.data
```

Let's do some basic tests. If the list has a length of 10 and we use m=0, then neither value error is raised, nor is the index error raised. We begin the while loop. rear_pointer is None, so counter is incremented. Counter is equal to m, so rear_pointer is set to head (which is the same as lead_pointer). Thus when we get to the end of the list, we will return the last element. We should probably raise a value error if m is not an integer at the top so the m<0 line doesn't throw an error. We should probably also check that the node passed through head has the necessary attributes of a singly linked list node. If m = 2, rear_pointer gets set to head when lead_pointer is head.next.next, so we will return the second-to-last element when lead_pointer reaches the tail.

Notes based on book solution:

- None

Problem: *List Flattening*. Start with a standard doubly linked list. Now imagine that in addition to next and previous pointers, each element has a child pointer, which may or may not point to a separate doubly linked list. These child lists may have a child of their own, which may have its own child, and so on. Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head and tail of the first level of the list. Each node is an instance of the following class:

```
class Node:
    def __init__(self, data):
        self.next = None
        self.prev = None
        self.child = None
        self.data = data
```

Robert Morgan
Dec. 2020, Jan. 2021

Solution

I would first want to know whether the final order mattered, since that would affect the ordering of the output. Let's assume it doesn't and then adapt the algorithm if necessary.

The multi-level data structure described in the problem is also referred to as an n-ary tree. Particularly since each node only has one possible child, this data structure can be thought of as a binary tree.

```
class Node:
    def __init__(self, data):
        self.next = None   # same as "right" in a BT node
        self.prev = None   # same as "parent" in a BT node
        self.child = None   # same as "left" in a BT node
        self.data = data
```

Therefore, one way we can flatten the data structure is traversing the data structure using either a breadth-first or depth-first search (depending on how the desired ordering of the output) and appending each element to a new array each time. However, this approach is spatially inefficient since copying the elements into a new array requires double the space.

To flatten the list without requiring additional space, we'll need to adjust the pointers instead of copying the elements. We can do this flattening slightly more efficiently by using a breadth-first search (ie. traverse the list with the next pointer, and every time a node has a child, load the child into a queue, then once you reach the tail pointer, set the next element to be the first element dequeued). This method only requires a lot of memory if the number of children in a particular level is huge, otherwise the space needed is much less than the full array.

The most efficient solution would not need a temporary data structure to store nodes in progress. Perhaps iterate through the list using the next pointer, and then whenever a node has a child, set the next pointer of tail to point to the child (and prev of child to point to tail), and then update the tail pointer to … Hmm. Updating the tail pointer kind of sucks because you have to iterate through all the next attributes of the child. Perhaps a depth first search is best, and each time you encounter a child set it to be the next attribute. The issue here is losing the connection to the next attribute while processing the information from the child. I think the intermediate data structure might be the best we can do.

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

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
```

```
        try:
            return self.queue.pop(0)
        except IndexError:
            raise IndexError("Queue is empty, nothing to dequeue")

    def is_empty(self):
        return len(self.queue) == 0

def flatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError("Cannot flatten empty list")

    queue = Queue()
    queue.enqueue(head)
    while not queue.is_empty():
        current = queue.dequeue()
        if current.child is not None:
            queue.enqueue(current.child)
        if current.next is not None:
            queue.enqueue(current.next)

        tail.next = current
        current.prev = tail
        tail = current
        del current.child
        del current.next
```

Notes based on book solution:

- Instead of loading into a queue, just stick the child at the end, and you'll eventually get to it to handle its children once your iteration reaches that point. This is O(n), simpler, and requires no additional space for an intermediate data structure

```
def flatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError("Cannot flatten empty list")

    current = head
    end_node = tail
    while current is not end_node:
        if current.child is not None:
            tail.next = current.child
            current.child.prev = tail
            current.child.next = None
            tail = current.child
        current = current.next
```

Problem: *List Unflattening*. After having flattened the list above, restore the original multi-level data structure.

Solution

We have the advantage that our approach never changed the child pointers of the nodes, so that is what we will have to use to reconstruct the multi-level list. Because we appended to the end, each time a node in the new list has a child, that indicates a new level. All we need to do is adjust the next / prev pointers and we should be good to go.

This will involve some recursion to explore the possible children of the child.

```
def unflatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError(“Cannot flatten empty list”)

    def explore_and_separate(start):
        if start is None:
            return

        if start.child is None:
            return

        else:
            start.child.prev = None  # Use None pointer to indicate new level
            start.child.prev.next = None
            explore_and_separate(start.child)

    current = head
    tail_updated = True # update tail after first child found, then use to terminate
    while current is not tail:
        if current.child is not None:
            if not tail_updated:  # Update the tail when the first child is found
                tail = current.child.prev
                tail_updated = True

            explore_and_separate(current.child)

        current = current.next
```

Notes based on book solution:

- None

Problem: *Null or Cycle*. You are given a linked list that is either null-terminated (acyclic) or ends in a cycle (cyclic). Write a function that takes a pointer to the head of the list and determines if the list is cyclic or acyclic. You may not modify the list in any way. Note that the data contained by a node might be identical to the data contained in another node, but that the two nodes are independent.

Solution

Robert Morgan
Dec. 2020, Jan. 2021

Since we cannot modify the list, we are not able to add a boolean `has_visited` flag to each element as we traverse the list. This makes me think we should copy the elements to a secondary data structure. Let's think about a hashset (something with constant time lookup). We cannot use the data as the keys, since two nodes can have the same data. We could use a counter to index the nodes, but we risk updating the counter incorrectly when a cycle does occur. Hashset might be out.

Obviously our solution should have the property that if it encounters a pointer to `None`, it stops and returns "acyclic" as the answer. The brute force approach would copy every node as it is encountered and add a boolean flag, however this is spatially inefficient.

A fun python-specific approach is to store the memory address of each node in a constant-time lookup data structure, then ask if the next node in question's memory address is in the array. Since we'd only be storing memory addresses, the overhead would be significantly lighter than copying the whole list.

```
def is_cycle(head):
    if head is None:
        raise ValueError("Null pointer passed to is_cycle")

    memory_addresses = set()
    current = head
    while current is not None:
        if id(current) in memory_addresses:
            return True
        else:
            memory_addresses.add(id(current))
        current = current.next
    return False
```

Notes based on book solution:

- You could compare the node to all its predecessors by starting at the head and iterating until you reach the current node. If any nodes are equal then you've found a cycle. The trouble here is that this approach is O(n^2).
- A clever solution is to use two pointers, and to increment them at different speeds. Therefore, if a cycle is present, the fast pointer will eventually equal the slow pointer. This approach is spatially efficient and O(n).

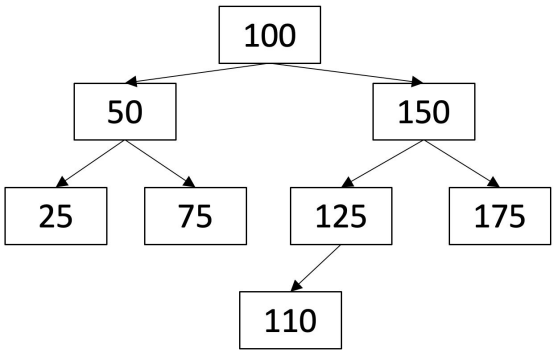
```
def is_cycle(head):
    if head is None:
        raise ValueError("Null pointer passed to is_cycle")

    fast_pointer = head
    slow_pointer = head
    try:
        while fast_pointer.next is not None:
            if fast_pointer is slow_pointer:
```

```
            return True
        fast_pointer = fast_pointer.next.next
        slow_pointer = slow_pointer.next

# Attribute error caused by end of list not having a next attribute, so acyclic
except AttributeError:
    return False

return False
```

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 4: Trees and Graphs

Problem: *Preorder Traversal*. Informally, a preorder traversal involves walking around the tree in a counter-clockwise manner starting at the root, sticking close to the edges, and printing out the nodes as you encounter them. For the tree to the right, the result is 100, 50, 25, 75, 150, 125, 110, 175. Perform a preorder traversal of a binary search tree, printing the value of each node.



Solution

In a preorder traversal, we visit the root, then visit the left subtree, then visit the right subtree. We can straightforwardly implement this traversal with a recursive function.

```
class BSTNode:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def __str__(self):
        return str(self.data)

def preorder_traversal(root):
    if root is None:
        return

    print(root)
    preorder_traversal(root.left)
    preorder_traversal(root.right)
```

Testing this solution on the example tree yields the output 100, 50, 25, 75, 150, 125, 110, 175 as intended.

Notes based on book solution:

- None

Problem: *Preorder Traversal, No Recursion*. Perform a preorder traversal of a binary search tree, printing the value of each node. This time, you may not use recursion.

Solution

Robert Morgan
Dec. 2020, Jan. 2021

Sure. Why not. A preorder traversal of a binary tree is the same as a depth-first search. Thus, we can load the root onto a stack, and while the stack is not empty pop and print, then add the children to the stack. The only trick here is to match the preorder behavior, we have to load the right child onto the stack first, such that the left child is always last-in-first-out.

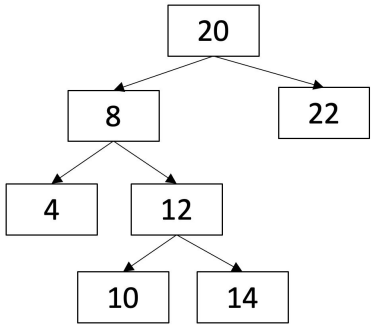
```
class Stack:
    # Stack.peek() is not needed for this example
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        return self.stack.pop()

    def is_empty(self):
        return len(self.stack) == 0

def preorder_traversal(root):
    stack = Stack()
    stack.push(root)
    while not stack.is_empty():
        current = stack.pop()
        print(current)
        if current.right is not None:
            stack.push(current.right)
        if current.left is not None:
            stack.push(current.left)
```

Testing on the example tree yields 100, 50, 25, 75, 150, 125, 110, 175 as intended.

Notes based on book solution:

- None

Problem: *Lowest Common Ancestor*. Given the value of two nodes in a binary search tree, find the lowest common ancestor. You may assume that both values already exist in the tree. For example, assume 4 and 14 are given as the two values for the tree to the right. The lowest common ancestor would be 8 because it's an ancestor to both 4 and 14 and there is no node lower on the tree that is an ancestor to both 4 and 14.



Solution

Unfortunately we cannot treat the tree as an array because it is not complete. This has the feel of a recursive problem where for a given subtree you ask "are both 4 and 14 in my left (or right)

subtree?" and if the answer is yes you ask the same question of your left (or right) child. Once you get to a point where the answer is no, meaning one of 4 and 14 is in the left child and one is in the right child, you have found the lowest common ancestor. This approach is somewhat inefficient because it essentially does log(n) depth first searches to verify that 4 and 14 are in the children, so maybe we can do better than reassessing the same nodes several times.

One property we did not use is the "search" property of the binary search tree (i.e. left child < root < right child). This may save us some time. Let's start at 20. Since both 4 and 14 are less than 20, we know they both must be in the left subtree, so we go to 8. Since 4 is less than 8 and 14 is greater than 8, we know 4 must be in the left tree and 14 must be in the right tree. Thus, we've found our common ancestor by only visiting two nodes. This approach is O(log(n)). Let's implement it.

```
def find_lowest_common_ancestor(root, value1, value2):
    assert value1 < value2
    if value1 < root.data and value2 < root.data:
        find_lowest_common_ancestor(root.left, value1, value2)
    elif value1 > root.data and value2 > root.data:
        find_lowest_common_ancestor(root.right, value1, value2)
    elif value1 < root.data and root.data < value2:
        return root.data
    else:
        raise RuntimeError("Invalid binary search tree passed as root")
```

Testing the example, we start at 20 and we execute the if clause. We then move to 8, and this time we execute the second elif clause, returning the correct answer of 8.

Notes based on book solution:

- Since we're not analyzing several branches and just going down the tree once, an iterative approach may be simpler that a recursive approach
- An iterative approach will also have fewer function calls, so in practice it may be faster than the recursive solution even though both are O(log(n))

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 5: Arrays and Strings

Problem: *First Non-repeated Character*. Write an efficient function to find the first non-repeated character in a string. For instance, the first non-repeated character in “total” is “o” and the first non-repeated character in “teeter” is “r”. Discuss the efficiency of your algorithm.

We’ll definitely have to scan every character in the string, so efficiency-wise we are bounded by O(n). Brute force would be to iterate through the string and for each character iterate through the remaining characters to determine whether the character appears again. This approach would be O(n^2).

```
def find_first_nonrepeated_char(word):
    for idx, char in enumerate(word):
        repeated = False
        for ch in word[idx:]:
            if char == ch:
                repeated = True
                break
        if not repeated:
            return char
    # return a flag if no non-repeated characters are found
    return 0
```

Slightly more memory efficient would be to iterate through the string and track both the indices characters in a hash table. This would be O(n), but would require additional space for the hash map. Although, the hashmap would only need as many keys as the number of unique characters in the word, so overall that’s not very much additional space. In Python, we can use an OrderedDict to also handle keeping the entries in the order in which they appear in the word. Let’s go with that.

```
from collections import OrderedDict

def find_first_nonrepeated_char(word):
    occur_dict = OrderedDict()
    for idx, char in enumerate(word):
        try:
            occur_dict[char] += 1
        except KeyError:
            occur_dict[char] = 1

    for char, occurrences in occur_dict.items():
        if occurrences == 1:
            return char
    # return a flag if no non-repeated characters are found
    return 0
```

Notes based on book solution:

- It would also be possible to use an array for the lookup, but this array will always be the size of the alphabet whereas the hashtable has the advantage of only needing to be the size of the number of unique characters in the word.

Problem: *Remove Specified Characters*. Write an efficient function that deletes characters from a string. For example, given a string of “Battle of the Vowels: Hawaii vs. Gronzy” and a removal argument of “aeiou”, the string should be altered to “Bttl f th Vwls: Hw vs. Grzny”. Justify any design decisions you make and discuss the efficiency of your solution.

So the brute for solution would be to iterate through the remove argument, and within that iteration, iterate through the string and remove any matches. This is O(n^2) and could certainly be improved.

I would load the remove argument into a hashtable, and then iterate through the string. If a string character is in the hashtable, then remove it from the string.

```
def remove_chars(string, remove):
    removal_map = {ch: None for ch in remove}
    out_string = []
    for ch in string:
        try:
            _ = removal_map[ch]
        except KeyError:
            out_string.append(ch)
    return ‘’.join(out_string)
```

In this solution, I’m choosing to append to a list and then join, rather than using string concatenation, because under the hood string concatenation with the “+” operator leads to a lot of unnecessary temporary variables. I’m also using a hash table for constant time lookup, and setting all the hash values to None to avoid unnecessary memory usage. The efficiency here is O(n + r) where n is the length of the string and r is the length of the removal argument.

Notes based on book solution:

- Building up a new string could become memory inefficient for long strings
- The proposal in the book suggests overwriting characters in the string as you iterate through it once. This works in C, but in Python strings do not support index-wise operations like insertion and deletion. Below is the book solution, which requires the extra step in Python of casting the string to a list

```
def remove_chars(string, remove):
    removal_map = {ch: None for ch in remove}
    string_arr = string.split(‘’)
    source_counter = 0
    destination_counter = 0
    while source_counter < len(string_arr):
        try:
            _ = removal_map[string_arr[source_counter]]
```

```
        except KeyError:
            string_arr[destination_counter] = ch
            destination_counter += 1
        finally:
            source_counter += 1
    return ‘’.join(string_arr[:destination_counter])
```

- Note that in Python, nothing is gained by this approach because we had to copy the string to an array, so we might as well have just used that memory to build up our string

Problem: *Reverse Words*. Write a function that reverses the order of the words in a string. For instance, your function should transform the string “Do or do not, there is no try.” to “try. no is there not, do or do”. Assume that all words are space delimited and treat punctuation the same as letters.

In Python, this is super easy and can be done using only standard built in functions.

```
def reverse_words(string):
    string_arr = string.split(‘ ‘)
    string_arr.reverse()
    return ‘ ‘.join(string_arr)
```

Let’s try to solve it without built in functions too, just in case that is asked.

Since we’ll have to scan the string for spaces, we’ll definitely be at least O(n) for our approach. Since this problem was written for C, we’ll use a list of characters for our string so that we have index-based operations available to us. The plan is to start a counter at the end of the string, iterate through the string backwards and decrement the counter as we go; every time we find a space, we put the chunk of the array between our counter and the end at the end of the array.

```
def reverse_words(string_arr):
    end = len(string_arr)
    back_counter = end
    counter = end - 1
    while counter >= 0:
        if string_arr[counter] == ‘ ‘:
            string_arr.extend(string_arr[counter:back_counter])
            back_counter = counter
        counter -= 1
    return string_arr[end:]
```

Notes based on book solution:

- A way to do this inplace is to iterate through the string twice. The first time, fully reverse it by swapping letters with their complements. Now the words will be in the correct order but the letters within the words will be backwards. The second time you iterate through, once you hit a space, stop, and swap all characters in that word, then continue. Technically this is still O(n) and does not require additional memory.

```
def reverse_string(string_arr, start, stop):
    while start < stop:
        string_arr[start], string_arr[stop] = string_arr[stop], string_arr[start]
        start += 1
        stop -= 1

def reverse_words(string_arr):
    reverse_string(string_arr, 0, len(string_arr)

    start_counter, stop_counter = 0, 0
    while stop_counter < len(string_arr):
        if string_arr[stop_counter] == ‘ ‘:
            reverse_string(string_arr, start_counter, stop_counter)
            start_counter = stop_counter
        stop_counter += 1
```

- Certainly more elegant, and creates the solution in place, but you need to make sure the string starts with a space such that the last word in the reversed string gets un-reversed

Problem: *Integer/String Conversions*. Write a function to convert an ASCII string to a signed integer, and write a function to convert a signed integer into an ascii string.

This question again feels weird for Python because builtin functions exist for type casting. The int() and str() functions already meet all the requirements of this problem. Regardless, let’s pretend those functions don’t exist.

To convert a string to an integer, we would want to first handle the sign, then we want to add the value of each digit to the result. This will be O(n).

```
def str_to_int(string):
    if string[0] == ‘-’:
        sign = -1
        end = 1
    elif string[0] == ‘+’:
        sign = 1
        end = 1
    else:
        sign = 1
        end = 0

    digit_map = {‘0’: 0, ‘1’: 1, ‘2’: 2, … ‘9’: 9}
    integer = 0
    power = 0
    start = len(string) - 1
    while start >= end:
        integer += digit_map[start] * 10**power
        power += 1
        start -= 1
```

```
    return sign * integer

def int_to_string(integer):
    sign = '-' if integer < 0 else ''
    exp = 1
    out_str = []
    zero = ord('0')
    abs_integer = abs(integer)
    while exp <= abs_integer:
        digit = (abs_integer // exp) % 10
        out_str.insert(0, chr(digit + zero))
        exp *= 10

    return sign + ''.join(out_str)
```

Both of these functions are O(n) where n is the number of digits in the integer / characters in the string. The only memory required in both cases is that of the final string / integer and the 10 key hashtable used for conversion.

Notes based on book solution:

- In the str_to_int function, you don't have to go backwards and track both the index and the exponent (the index already carries all the information that the exponent needs anyways). Instead, start from the front and everytime you move a place, multiply your number by 10. This optimization is called *Homer's Rule*.

```
def str_to_int(string):
    if string[0] == '-':
        sign = -1
        start = 1
    elif string[0] == '+':
        sign = 1
        start = 1
    else:
        sign = 1
        start = 0

    zero = ord('0')
    integer = 0
    for digit in string[start:]:
        integer *= 10
        integer += ord(digit) - zero
    return sign * integer
```

- I've revised the int_to_str function to use ord and chr functions instead of a hashtable, but the algorithm is the same as the solution

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 6: Recursion

Problem: *Binary Search*. Implement a function to perform a binary search on a sorted array of integers to find the index of a given integer. Comment on the efficiency of this search and compare it to other search methods.

Solution

For binary search, we start in the middle and ask “is this element greater than or less than my element”, and then move to the middle of that half. We repeat this process recursively until an element is found or the sub-array in question is empty.

```
def binary_search(arr, element):
    size = len(arr)
    if size == 0:
        raise ValueError(f“{element} not in array”)

    midpoint = size // 2
    if arr[midpoint] > element:
        return binary_search(arr[:midpoint], element)
    elif arr[midpoint] < element:
        return binary_search(arr[midpoint + 1:], element)
    else:
        return midpoint
```

Tests. Empty array and element not in array will raise a ValueError. For binary_search([1, 2, 3, 4], 2), we get midpoint = 2, 2 < 3, so we run binary_search([1, 2], 2), midpoint = 1, 2==2, so we return 1.

Binary search on a sorted array is O(log(n)), and is faster than a linear search in cases where the element is not right at the beginning of a very long list.

Notes:

- It might be more efficient to utilize a while loop instead of recursion. Each time through the loop you can update the limits of the sub-array to search to avoid several function calls

Problem: *Permutations of a String*. Implement a function that prints all possible orderings of the characters in a string. For instance, given “hat”, you should print all 6 permutations. Given “aaa”, you should print “aaa” six times (characters are treated as unique). The permutations can print in any order.

Solution

I believe there is built in functionality for this sort of thing using the ittertools module in the standard library. Nonetheless, let’s go for it.

Robert Morgan
Dec. 2020, Jan. 2021

If a string is length 1 or empty, we just return it. If a string is length 2 we return it and the swap of its letters. For a length three string, we can think of it as a 3 combinations of a length 1 string and a length 2 string. For a length 4 string, we can think of it as 4 combinations of a length 3 string and a length 1 string. This feels like a recursive approach. Efficiency-wise, I think we're bounded by factorial order, since we have to produce n! permutations.

One problem is that we don't want to build up a list of all permutations, since that could get huge. We just want to print them. Therefore, everytime we have a permutation, we should print it and move on, so let's adjust the recursive approach to do this. Actually, since we're stuck with factorial order anyways, loops would be just as fast, if not faster, as a recursive function

```
def print_permutations(string_arr, curr_idx=0):
    if curr_idx == len(string_arr) -1:
        print(‘’.join(string_arr))

    for i in range(curr_idx, len(string_arr)):
        string_arr[curr_idx], string_arr[i] = string_arr[i], string_arr[curr_idx]
        print_permutations(string_arr, curr_idx + 1)
        string_arr[curr_idx], string_arr[i] = string_arr[i], string_arr[curr_idx]
```

*Help from https://www.techiedelight.com/find-all-permutations-string-python/

Problem: *Combinations of a String*. Implement a function that prints all possible combinations of the characters in a string. These combinations range in length from one to the length of the string, but two combinations that differ only in the ordering of their characters are the same combination.

This time around, instead of swapping elements, we want to find all the possible sub arrays. Let's step through an example to deduce an algorithm. Consider the string "abc". We can start by finding all the sub arrays with length = 3, which is just the original array. Then we go down to all the sub arrays with length =2. There are three arrays in this case (3C2). They are found by separating the first element and returning the other two, then separating the second element and returning the other two, and the same for the third element. Then we move to length 1 arrays, for which we remove len(string)-counter elements each time and find all the substrings.

So we have an approach for how to iterate through the full string, now we just need a way to find all the substrings of a given length from an original. Note that there is a symmetry here, when we find a subarray of length size-n, the other array is part of the solution for the set of subarrays of length n.

I'm picturing splitting the array at a certain point, then shifting the elements in the array by one, moving the last element to the front. Always print both sub strings. This won't get all the sub strings, better to develop a recursive approach

```
def combinations(l):
```

```
    if len(l) > 0:
        result = combinations(l[:-1])
        return result + [c + [l[-1]] for c in result]
    else:
        return [[]]

def print_combinations(string):
    for combo in combinations(list(string)):
        print(''.join(combo))
```

*With help from https://stackoverflow.com/questions/65181109/recursively-find-all-combinations-of-list

Problem: *Telephone Words.* Write a function that takes a seven-digit telephone number and prints out all of the possible "words" of combinations of letters that can represent the given number using a standard telephone keypad. For '1' and '0', just use '1' and '0'. Assume your input is an array of seven numbers.

Solution

So for each digit, we will have either one possibility (0,1), three possibilities (2, 3, 4, 5, 6, 8, 9), or 4 possibilities (7). The order of the digits stays the same, we just have to cycle through each of the possibilities for each digit. It's not pretty, but we know we'll always have seven digits, so we could set up seven nested for loops.

```
def phone_words(number_arr):
    digit_map = {0: ['0'], 1: ['1'], 2:['a', 'b', 'c'], …}
    for d1 in digit_map[number_arr[0]]:
        for d2 in digit_map[number_arr[1]]:
            …
            for d7 in digit_map[number_arr[6]]:
                print(''.join([d1, d2, d3, d4, d5, d6, d7]))
```

Maybe sometime later in life we can be smarter and come up with a recursive solution.

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 7: Other Programming Topics

Problem: *Eighth of a Circle*. Write a function that draws the upper eight of a circle centered at (0,0) with a given radius, where the upper eighth is defined as pi/2 to pi/4. To draw the circle, you have at your disposal a function that fills in a single pixel given an (x,y) coordinate.

Solution

The main issue here will be rounding. Let's assume that the coordinate points to the center of the pixel, and that the given radius is in pixel coordinates. One approach is to step the angle through small increments, where the increments need to be chosen small enough such that no gaps are left between pixels. The increments would be a function of the input radius. Specifically, to draw a quarter of a circle, we need the same number of pixels as the length of the radius, so for one eighth we'll need the radius divided by 2.

```
import math

def draw_eighth_of_circle(radius):
    angle_increment = math.pi / 4 / (radius / 2)
    angle = math.pi / 4
    while angle < math.pi / 2:
        x = int(round(math.cos(angle) * radius))
        y = int(round(math.sin(angle) * radius))
        fill_pixel(x, y)
        angle += angle_increment
```

Notes based on book solution:

- None

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 8: Counting, Measuring, and Ordering Puzzles

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 9: Graphical and Spatial Puzzles

Robert Morgan
Dec. 2020, Jan. 2021

Online Google Interview Questions

The 15 Most Asked Questions in a Google Interview

From an online article: [The 15 most asked questions in a Google Interview]

Problem: *Delete Node with Given Key*. You are given the head of a linked list and a key. You have to delete the node that contains the given key.

Solution

First I would like to know whether this is a singly linked list or a doubly linked list. I will proceed assuming singly linked, since it's slightly harder than deleting from a doubly linked list. The trick is connecting the previous node to the following node, so in a doubly linked list you have a pointer to the previous node, but in a singly linked list you don't have that luxury.

```
def delete_node_with_key(head, key):
    prev = None
    current = head
    while current.key != key:
        prev = current
        current = current.next

    prev.next = current.next
    del current
```

Notes based on online solution:

- Account for the possibility of the key not being in the list

Problem: *Copy Linked List with Arbitrary Pointer*. You are given a linked list where the node has two pointers. The first is the regular "next" pointer. The second pointer is called "arbitrary_pointer" and it can point to any node in the linked list. Your job is to write code to make a deep copy of the given linked list. Here, deep copy means that any operations on the original list (inserting, modifying, or removing) should not affect the copied list.

Solution

The issue here is that if you were to iterate through the list and copy nodes and pointers, you might get to the situation where the "arbitrary_pointer" points to a node that has yet to be copied, so it would become a null pointer. To get around this, we might copy the values and next pointers first, and then copy the arbitrary pointers with a second iteration. This is O(n), but requires visiting each node twice.

Come to think of it, copying next pointers kind of sucks too because they won't have anything to point to until the following node is copied. For the deep copy part of the problem, we'll need to

create new nodes and set their pointers to other new nodes, so that will definitely be a part of the solution.

Another issue, setting the arbitrary_pointer requires you to have the node it's pointing to on hand, so even if the copy already exists, you still need an efficient way to find the copy. This lookup problem makes me want to use a hashmap to store the copies while their pointers are being set. Let's give that a try.

Here I'm going to assume the data values in the linked list are unique, such that I can use them as keys in a dictionary. If they are not unique, I would just use id(node) as the key in the dictionary. Actually, let's just do that for completeness.

I would also like to know about the data stored in the list nodes, to determine if making a deep copy of that data is an essential part of the problem. It seems like the point is to properly copy the pointers, so I will assume the data is strings or integers such that a copy and deep copy are the same.

```
class ListNode:
    def __init__(self, data):
        self.next = None
        self.arbitrary_pointer = None
        self.data = data

def copy_list_with_arbitrary_pointer(head):
    new_nodes = {}
    current = head
    while current is not None:
        key = id(current)
        # Copy the current node
        try:
            existing_copy = new_nodes[key]
        except KeyError:
            new_nodes[key] = ListNode(current.data)
            existing_copy = new_nodes[key]

        # Copy the next pointer
        if current.next is not None:
            next_key = id(current.next)
            try:
                existing_next = new_nodes[next_key]
            except KeyError:
                new_nodes[next_key] = ListNode(current.next.data)
                existing_next = new_nodes[next_key]
            existing_copy.next = existing_next

        # Copy the arbitrary pointer
        if current.arbitrary_pointer is not None:
            arbitrary_key = id(current.arbitrary_pointer)
```

```
            try:
                existing_arbitrary = new_nodes[arbitrary_key]
            except KeyError:
                new_nodes[arbitrary_key] = ListNode(current.arbitrary_pointer.data)
                existing_arbitrary = new_nodes[arbitrary_key]
            existing_copy.arbitrary_pointer = existing_arbitrary

    # Return the head of the copied linked list
    return new_nodes[id(head)]
```

Here I have performed a deep copy by creating new ListNode instances for each node, setting the data attribute appropriately, and setting pointers only to copied nodes. This solution is also O(n) since all the nodes are only visited once, and since lookup in the dictionary is O(1).

Notes based on online solution:

- Could just use the node itself in the hashtable instead of its id but the concept is the same

Problem: *Mirror Binary Trees*. Given the root node of a binary tree, swap the left and right children for each node.

Solution

I swear if I get asked to invert a binary tree…. It's not hard, start at the root and recursively interchange the children of each node you visit.

```
def mirror_binary_tree(root):
    # Base condition
    if root is None:
        return

    # Swap
    root.left, root.right = root.right, root.left

    # Recurse
    mirror_binary_tree(root.left)
    mirror_binary_tree(root.right)
```

Notes based on online solution:

- My solution was preordered; theirs was postordered. I don't think it matters.

Problem: *Check if Two Binary Trees are Identical*. Given the roots of two binary trees, determine if these trees are identical or not.

Solution

Robert Morgan
Dec. 2020, Jan. 2021

An O(n) solution here is to do either a depth first or breadth first traversal of both trees simultaneously, and check that the roots are always equal. No need to check the pointers to the children because if they are identical, the children get visited in the same order.

Actually, this doesn't account for the interchange of a left leaf with a right leaf for a node with only one child. The traversal order could be the same. We could either check the pointers to the children, but the simpler way is to do an inorder traversal, since we'll be able to tell if a child is left or right based on its ordering with respect to the root.

```
def are_identical(root1, root2):
    if root1 is None and root2 is not None:
        return False
    elif root2 is None and root1 is not None:
        return False
    elif root1 is None and root2 is None:
        return True

    left = are_identical(root1.left, root2.left)
    root = root1.data != root2.data
    right = are_identical(root1.right, root2.right)

    return left and root and right
```

Problem: Sum of two values

```
def sum_of_two_values(arr, sum_value):
    hashset = set()
    for value in arr:
        if sum_value - value in hashset:
            return True
        hashset.add(sum_value - value)

    return False
```

Problem: Move all zeros to the left.

```
def move_zeros_left(arr):
    # a la insertion_sort
    counter = 0
    while counter < len(arr):
        if arr[counter] == 0:
            arr.insert(0, arr[counter])
        counter += 1
```

I don't like it because it feels like every insertion can require at most n element shifts. There must be a better way

Problem: String Segmentation.

Robert Morgan
Dec. 2020, Jan. 2021

```
def segment(string):

    for idx in range(len(string)):
        first_word = string[:idx]
        if dictionary.contains(first_word):
            second_word = string[idx:]
            if (len(second_word) == 0 or
                dictionary.contains(second_word) or
                segment(second_word)):
                return True
    return False
```

Problem: Find all palindrome strings.

```
def find_palindromes(string, center_idx):
    l_counter, r_counter = center_idx, center_idx
    palindromes = []
    while l_counter >= 0 or r_counter < len(string):
        if string[l_counter] == string[r_counter]:
            palindromes.append(string[l_counter:r_counter+1])
            l_counter -= 1
            r_counter += 1
        else:
            return palindromes

def is_a_bunch_of_palindromes(string):
    l_counter = 0
    r_counter = 2
    all_palindromes = []
    while r_counter < len(string):
        if string[l_counter] == string[r_counter]:
            all_palindromes.extend(find_palindromes(string, l_counter + 1))

        l_counter += 1
        r_counter += 1

    return all_palindromes
```

Notes:

- You forgot even-length palindromes

Problem: *find the high and low index*. Given a sorted array of integers, return the low and high index of the given key. Return -1 if not found. The array length can be in the millions with many duplicates.

Binary search, then linear search.

```
def binary_search(arr, element):
```

```
    size = len(arr)
    if size == 0:
        raise ValueError(f“{element} not in array”)

    midpoint = size // 2
    if arr[midpoint] > element:
        return binary_search(arr[:midpoint], element)
    elif arr[midpoint] < element:
        return binary_search(arr[midpoint + 1:], element)
    else:
        return midpoint

def find_high_low(arr, element):
    index = binary_search(arr, element)
    low = index
    while arr[low] == element or low > 0:
        low -= 1
    while arr[index] == element or index < len(arr):
        index += 1
    return low, index
```

No, binary search for high, then binary search for low. O(2 log(n)) better than O(n + log(n))

Problem: *Merge Overlapping Intervals*. You are given an array (list) of interval pairs as input where each interval has a start and end timestamp. The input array is sorted by starting timestamps. You are required to merge overlapping intervals and return output array (list).

Proceed linearly through the list.

```
def merge_overlaps(arr):
    output_arr = []
    for idx, timestamp in enumerate(arr):
        if idx == 0:
            start = arr[idx][0]
            end = arr[idx][1]
            continue

        if timestamp[0] <= end and timestamp[1] > end:
            end = timestamp[1]
        elif end < timestamp[0]:
            output_arr.append((start, end))
            start = timestamp[0]
            end = timestamp[1]

        if idx == len(arr) - 1:
            output_arr.append((start, end))


    return output_arr
```

Robert Morgan
Dec. 2020, Jan. 2021

Problem: *Valid Braces*. Given a number N, find all valid brace combinations. 1 = {}, 2 = {}{}, {{}}, etc.

```
def valid_braces(n):
    assert n >= 1, “N must be greater than or equal to 1”
    if n == 1:
        return ‘{}’
    return [‘{‘ + x + ‘}’ for x in valid_braces(n-1)]
```

https://apprenticedeveloper.com/interview-coding-problem-strings-arrays/

Robert Morgan
Dec. 2020, Jan. 2021

Cracking the Coding Interview

Book by Gayle Laakmann

Chapter 1: Arrays and Strings

Problem 1.1: *Unique Characters*. Write an algorithm to determine if all the characters in a string are unique. What if you cannot use additional data structures.

Solution

Easy way is to use a set and for each character add it to the set and ask whether it's in the set. This will be O(n) since lookup in the set will be O(1). If possible, could we do it without the extra memory overhead? In the unique case, the set will contain all the characters, so we will require double the memory.

To handle the memory more efficiently, we could utilize the size of the ASCII alphabet to help us. Since there are only 256 characters, we could use an array of length 256 instead of a set. I would argue though, that if we know the size of the alphabet, no string with more than 256 characters will be unique, so the array method will always require a memory overhead of a length-256 array, but the set method will be more memory efficient if the string is not unique.

If we cannot use additional data structures, then we are forced to do element-wise comparisons, which will be an O(n^2) run time. If we are allowed to destroy the string, we can do a modified insertion sort in O(n log(n)) time (to avoid taking up extra space) and at each insertion check whether two characters are equal.

Problem 1.2: *String Reversal*. Write code to reverse a C-style string (string with a null character at the end). For Python, assume a new-line character.

Solution

Here I propose to swap each element with its complement, perhaps using two counters, and stopping the iteration when the counters reach each other.

```
def reverse(string):
    l_counter = 0
    r_counter = len(string) - 2 # to avoid the '\n' character
    while l_counter < r_counter:
        string[r_counter], string[l_counter] = string[l_counter], string[r_counter]
        l_counter += 1
        r_counter -= 1
```

Now in Python, we cannot do index-based assignment in strings. Our only choice will be to create a new string. In this case, it's more memory efficient to append to a list instead of performing several string concatenations, since string concatenation requires temporary variables prior to assignment.

```
def reverse(string):
    out_str = []
    for ch in string[0:-1]:
        out_str.insert(0, ch)
    return ‘’.join(out_str) + string[-1]
```

Problem 1.3: *Remove Duplicate Characters*. Design an algorithm to remove duplicate characters in an string-array without using any additional data structures to copy the array.

Solution

So here we're not allowed to use any sort of hashset to track the characters that have already appeared. We also cannot use something like counting sort where we track the occurrences of each. We'll have to go through the array and compare each character to all the others and do it in O(n^2) time. If we can jumble the returned string, we can do it a la modified insertion sort in O(n log(n)).

If we're allowed a single variable of fixed memory size, I would use an array of length = len(alphabet) and count occurrences. Still, a set will outperform an array.

```
def remove_dup_chars(string_arr):
    letters = set()
    for idx, ch in enumerate(string_arr):
        if ch not in letters:
            letters.add(ch)
            string_arr[idx] = ‘’
    return ‘’.join(string_arr)
```

Problem 1.4: *Anagrams*. Write a method to determine if two strings are anagrams.

Solution

To be anagrams, the words must contain all of the same letters. I would iterate through the first string and build up a hashtable with keys of the letters and values of the counts. Then I would go through the other string and decrease the counts in the hashtable. If I am left with any nonzero counts or the second array produces a key error, the strings are not anagrams.

```
def are_anagrams(string_1, string_2):
    ht = dict()
    for ch in string_1:
        if ch in ht:
            ht[ch] += 1
```

```
        else:
            ht[ch] = 0

    for ch in string_2:
        try:
            ht[ch] -= 1
            if ht[ch] == 0:
            del ht[ch]
        except KeyError:
            return False

    return len(ht) == 0
```

If we can modify the strings (arrays) inplace and cannot utilize extra memory, insertion sort each one, and determine if they are equal for an O(n log(n)) solution.

Problem 1.5: *Replacement*. Write a method to replace all the spaces in a string (array) with '%20'.

Pythonically, this is straightforward: `string.replace(' ', '%20')`

If we want to solve the problem without the builtin method, while still treating the string as an array of characters, the problem is now to put in a string of length 3 where every space is. I guess we can do this linearly.

```
def replace(string_arr: list):
    counter = 0
    stop = len(string_arr)
    while counter < stop:
        if string_arr[counter] == ' ':
            string_arr.insert(counter, '%')
            string_arr.insert(counter + 1, '2')
            string_arr.insert(counter + 2, '0')
            counter += 2
            stop += 2
        counter += 1

    return string_arr
```

Problem 1.6: *Matrix Rotation*. Given an NxN matrix representation of an image, write a function to rotate the image by 90 degrees. Do the rotation inplace if possible.

Solution

The non-inplace solution would be to copy each element to its rotated location in a new matrix. We can do that inplace, we just have to do a simultaneous four-element swap.

```
def rotate(image):
```

```
    levels = len(image) // 2
    counter = 0
    while counter < levels:
        (image[counter][counter], image[counter][counter-1],
         image[counter-1][counter], image[counter-1][counter-1] =
         image[counter][counter-1], image[counter-1][counter-1],
         image[counter][counter], image[counter-1][counter])
        counter += 1
```

Problem 1.7: *Matrix Zeroing*. Write an algorithm such that if an element in a row or column is zero, its entire row and column are set to 0.

We don't want to iterate through the matrix if possible. This has the feel of some sort of weird matrix multiplication. In Python, we can use numpy to find the locations of all the zeros easily, but this will involve utilizing additional memory. Let's try to do it inplace.

In the brute force approach, we have O(n^2) where n is the dimensionality of the matrix, so since there are n^2 elements in the matrix, one could argue that the brute force approach is linear with the number of elements.

The problem here is that the zeros we set will trigger the iteration later on, and will cause more rows / columns to be set to zero. We can get around this by giving up on inplace manipulation, since storing the locations of all the zeros (in the zero-matrix case) could be as large as the original matrix. We don't need the full locations though, just the unique row and column indices are enough.

```
def zero_matrix(arr):
    rows = set()
    cols = set()
    for ii in range(len(arr)):
        for jj in range(len(arr)):
            if arr[ii][jj] == 0:
                rows.add(ii)
                cols.add(jj)

    for ii in range(len(arr)):
        for jj in range(len(arr)):
            if ii in rows or jj in cols:
                arr[ii][jj] = 0
```

Problem 1.8: *String Rotation*. Assume you have a method isSubstring which checks if one word is a substring of another. Given two strings, s1 and s2, write code to check if s2 is a rotation of s1 using only one call to isSubstring (i.e., "waterbottle" is a rotation of "erbottlewat").

Solution

Robert Morgan
Dec. 2020, Jan. 2021

First thought, start a counter in s1 and once the letter reached in s1 is the same as the first letter in s2 start comparing letters. This fails if the letter appears multiple times, though.

You could start at the beginning of s1 and for each consecutive index ask if you have a substring of s2. If a word is a rotation, you'll get one or more yeses before a no (where the rotation happens). At the point of the first no, you could switch to just asking from your counter to the end of s1. If you get any more nos, it's not a rotation. If you reach the end of s1 with all yesses, it's a rotation. This approach is O(n), assuming the complexity of isSubstring is O(1).

```
def is_rotation(s1, s2):
    if len(s1) != len(s2): return False

    counter = 1
    while counter < len(s1):
        if isSubstring(s1[:counter], s2):
            counter += 1
        else:
            if counter == 1:
                return False
            else:
                counter -= 1
                break
    else:
        return True

    return s1[counter:] + s1[:counter] == s2
```

Let's test this approach with the "waterbottle" example. The steps are

1. Is "w" a substring of "erbottlewat"? yes, counter 1->2
2. Is "wa" a substring of "erbottlewat"? yes, counter 2->3
3. Is "wat" a substring of "erbottlewat"? yes, counter 3->4
4. Is "wate" a substring of "erbottlewat"? no. Is counter == 1? no. counter -> 3. Break.
5. Is s1[3:] + s1[:3] == s2? ("erbottle" + "wat" == "erbottlewat")? yes. return True

Let's test this with a non-rotation: s1 = "abc", s2 = "bcd"

1. Is "a" a substring of "bcd"? No. Is counter == 1? yes. return False

Let's test this with a non-rotation that makes it beyond the first letter: s1 = "bcde", s2 = "abcd"

2. Is "b" a substring of "abcd"? Yes. counter -> 2
3. Is "bc" a substring of "abcd"? Yes. counter -> 3
4. Is "bcd" a substring of "abcd"? Yes. counter -> 4
5. Is "bcde" a substring of "abcd"? No. is counter == 1? No. counter -> 3. Break
6. Is "e" + "bcd" a substring of "abcd"? No. return False.

This function will return True if both s1 and s2 are empty via the else clause of the while loop. Overall looks good and efficient!

Robert Morgan
Dec. 2020, Jan. 2021

Notes based on book solution:

- Could just concatenate s1 with itself and ask if s2 is a substring, and vice versa

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 2: Linked Lists

Problem 2.1: *Linked List Deletion*. Write code to remove duplicates from an unsorted linked list. How would you solve the problem without the use of a temporary data structure?

With a temporary data structure, you can proceed linearly through the list and store the node values in a hashset. Each time you encounter a node, if it exists in the hashset, delete the node by connecting the previous node with the next node (and vice versa if the list is doubly linked).

```
def delete_duplicates_linked_list(head):
    lookup = set()
    prev = None
    current = head
    while current is not None:
        if current.data in lookup:
            prev.next = current.next
            # if doubly linked list…
            # prev.next.prev = prev
            del current
            current = prev.next
        else:
            lookup.add(current.data)
            current = current.next
```

Without a temporary data structure, for each node, you will have to start at head and check all the nodes until the current node. This will be O(n^2).

Problem 2.2: *Nth-to-last*. Implement an algorithm to find the nth to last element of a singly linked list.

The trick is to use two counters, but start the second counter once the first counter has reached the nth element. Then, when the front counter reaches the end, the second counter will be at the nth to last element.

```
def nth_to_last(head, n):
    counter = 0
    lead_node = head
    while lead_node is not None:
        if counter == n:
            rear_node = head
        elif counter > n:
            rear_node = rear_node.next
        counter += 1

        lead_node = lead_node.next

    if counter < n:
        raise IndexError(“Not enough elements in linked list”)
```

```
    else:
        return rear_node.data
```

Problem 2.3: *Node Deletion with a Twist*. Implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node.

So the hard part here is figuring out how to set the next pointer from the previous element. Rather than remove the node and adjust the pointer, we can copy the data from all the following node left one node, thus replacing the node to delete, and we will set the final node in the list to None.

```
def delete_node(node):
    current = node
    while current.next is not None:
        current.data = current.next.data
        current = current.next
    current = None
```

Notes based on book solution:

- You don't have to shift all the nodes, you can just delete the next node the standard way because you will have access to the previous node

Problem 2.4: *Addition with linked lists*. You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

We can do this in linear order by mirroring the process of addition-by-hand.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def add_lists(head_1, head_2):
    if head_1 is None:
        return head_2
    if head_2 is None:
        return head_1

    curr_1, curr_2 = head_1, head_2
    result_head = Node(None)
    result = result_head
    carry_over = 0
    while curr_1 is not None or curr_2 is not None or carry_over != 0:
        # Account for zeros
        curr_1_data = 0 if curr_1 is None else curr_1.data
        curr_2_data = 0 if curr_2 is None else curr_2.data
```

```
        # Do the math
        val = curr_1_data + curr_2_data
        rem = val % 10
        # Account for double carry_over
        real_value = rem + carry_over
        result.next = Node(real_value % 10)
        carry_over = val // 10 + real_value // 10

        # Increment, accounting for different length lists
        result = result.next
        try:
            curr_1 = curr_1.next
        except AttributeError:
            pass
        try:
            curr_2 = curr_2.next
        except AttributeError:
            pass

    return result_head.next
```

Problem 2.5: *Linked List Cycle Detection*. Given a linked list with a loop, return the start of the loop.

```
def find_cycle_node(head):
    if head is None:
        return None

    current = head
    while current is not None:
        if hasattr(current, “visited”):
            return current
        current.visited = None

    return None
```

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 3: Stacks and Queues

Problem 3.1: *Three stacks*. Describe how you could use a single array to implement three stacks.

First three elements contain the index of where the tp element in a stack is. Each stack stores its elements in every third index. When you push and pop, you interact with the index in the first three elements of the array.

```
arr = [3, 4, 5, None, None, None]

class Stack:
    def __init__(self, start_idx):
        self.start_idx = start_idx

    def push(self, item):
        next_idx = arr[self.start_idx]
        arr[next_idx] = item
        arr[self.start_idx] += 3
        if len(arr) < arr[self.start_idx]:
            arr.extend([None, None, None])

    def pop(self):
        arr[self.start_idx] -= 3
        item = arr[self.start_idx]
        arr[self.start_idx] = None
        return item

stack_1 = Stack(0)
stack_2 = Stack(1)
stack_3 = Stack(2)
```

Problem 3.2: *Min Stack*. How would you design a stack that can perform its basic functions, but also return the minimum value in the stack in O(1) time?

Solution

If you implement the stack using a linked list, you could store an attribute in the head node that contains the minimum value, and allow that attribute to be updated every time you pop or push.

There is one special case where you will be somewhat inefficient, and that is if you pop the minimum value. You will have to iterate through the linked list to find the new minimum, which will be O(n). However, the actual minimum function itself will be O(1) most of the time.

```
class StackNode:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
        self.minimum = None

class MinStack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = StackNode(data)
        if self.top is None:
            self.top = new_node
            self.top.minimum = data
        else:
            minimum = min(self.top.minimum, data)
            new_node.next = self.top.next
            self.top = new_node
            self.top.minimum = minimum

    def pop(self):
        data_to_pop = self.top.data
        if data_to_pop == self.top.minimum:
            new_min = self.top.next.data
            current = self.top.next
            while current is not None:
                if current.data < new_min:
                    new_min = current.data
                current = current.next
        else:
            new_min = self.top.minimum

        self.top = self.top.next
        self.top.minimum = new_min
        return data_to_pop

    def minimum(self):
        return self.top.minimum
```

Notes based on the book solution:

- Alternatively, you could keep a second stack of just the minimum values; only push onto that stack when there is a new minimum. That way, when you pop the minimum from the main stack, the new minimum is next up in the minimum stack.

Problem 3.3: *Stack of Plates*. Imagine a literal stack of plates. If the stack gets too high, it might topple. Therefore, in real line, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetofStacks should be composed of several stacks, and should create a new stack once the previous one exceeds capacity. SetofStacks.push() and SetofStacks.pop() should behave identically to a single stack.

Solution

Robert Morgan
Dec. 2020, Jan. 2021

First of all, we need to count the number of items in the stack whenever push and pop are called, otherwise, we'd have to iterate through the stack to count. So SetOfStacks will need to have a first_stack_count attribute.

Perhaps we could use a dynamic array (Python list) to index the different stacks, which could either be implemented via a dynamic array or a linked list. Let's do dynamic arrays all around.

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

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        self.stack.pop()

    def peek(self):
        return self.stack[-1]

class SetOfStacks:
    def __init__(self, capacity):
        self.first_stack_count = 0
        self.capacity = 0
        self.stacks = [Stack()]

    def push(self, item):
        if self.first_stack_count == self.capacity:
            self.stacks.append(Stack())
            self.first_stack_count = 0
        self.stacks[-1].push(item)

    def pop(self):
        try:
            popped_item = self.stacks[-1].pop()
            self.first_stack_count = 0
            return popped_item
        except IndexError:
            del self.stacks[-1]
            try:
                self.first_stack_count = self.capacity - 1
                return self.stacks[-1].pop()
            except IndexError:
                raise IndexError("SetOfStacks is empty")
```

Follow-up problem: implement a pop_at_index() function to access a particular sub stack.

```
    def pop_at_index(self, index):
```

```
        try:
            # Note that my stacks start at the end of the list
            selected_stack = self.stacks[-1 * index]
        except IndexError:
            raise IndexError("Not enough stacks in SetOfStacks")

        popped_item = selected_stack.pop()
        # Need to move one item over from each previous stack
        current = -1 * index
        while current < -1:
            self.stacks[current].push(self.stacks[current+1].pop())
            current += 1
        self.first_stack_count -= 1

        return popped_item
```

Problem 3.4: *Towers of Hanoi*. Use stacks to solve the problem of moving N disks from the first rod to the final rod. You can only move one disk at a time, and you cannot place a larger disk atop a smaller disk.

Solution

Model each rod as a stack. Develop an algorithm for which stack to pop and push from under different conditions. Programming-wise, it's straightforward to implement 3 stacks and pop and push elements, and there are no real tricks (you just have to try a couple iterations and deduce the algorithm), so we'll move on.

Problem 3.5: *Queue*. Implement a queue using two stacks.

Solution

We need to set up the two stacks to be first in first out. We can achieve this by using one of the two stacks to reverse the order of the elements each time an item is pushed into the queue. This approach is O(n) dequeue, so there's a reason it's not done this way.

Rather than reversing the stacks twice each time, which would be inefficient if we wanted to pop twice in a row, we could determine whether the stacks were in "pop" mode or "push" mode, and only reverse when needed.

```
class Queue:
    def __init__(self):
        self.current_stack = Stack()
        self.backup_stack = Stack()

    def enqueue(self, item):
        self.current_stack.push(item)

    def dequeue(self):
```

```
        while not self.current_stack.is_empty():
            self.backup_stack.push(self.current_stack.pop())
        item = self.backup_stack.pop()
        self.backup_stack, self.current_stack = (
            self.current_stack, self.backup_stack)
        return item
```

Notes based on book solution:

- Rather than reversing the stacks twice each time, which would be inefficient if we wanted to pop twice in a row, we could determine whether the stacks were in “pop” mode or “push” mode, and only reverse when needed.

Problem 3.6: *Stack sorting*. Write a program to sort a stack in ascending order. You should not make any assumptions about how the stack is implemented. The following are the only functions that should be used to write this program: push | pop | peek | is_empty.

Solution

First though is to pop each item out and push each popped item into a max heap. Then we could pop from the max heap and push onto the stack, resulting in the stack being ordered in ascending order. This approach is O(n log(n)), so that’s probably the best we can do for a sorting algorithm. Technically the heap methods are also called pop and push, so this solution should be allowed ;).

If we’re not allowed to use an additional non-stack data structure, we’ll have to create more stacks. What we want to avoid is having to dig to the bottom of one of the other stacks to determine the maximum value. Thus, we should probably track what the maximum value of each stack is in the top node with an additional variable. Basically, by checking the maximum values, we determine if it’s possible to put an element at the top of a stack in sorted order, and if not we need to create a new stack.

This might save some efficiency, but the book just wants an O(n^2) approach, so we’ll just use one additional stack and pop as many items as needed until we find the right place to put the item of interest (a la insertion sort).

```
def stack_insertion_sort(stack):
    extra_stack = Stack()
    while not stack.is_empty():
        next_item = stack.pop()
        while not extra_stack.is_empty() and extra_stack.peek() > next_item:
            stack.push(extra_stack.pop())
        extra_stack.push(next_item)
    return extra_stack
```

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 4: Trees and Graphs

Problem 4.1: *Balanced Trees*. Implement a function to determine if a tree is balanced.

Solution

We need a way to calculate the height of each leaf node, which means we need to access each leaf node. We don't need to track all the heights, we just need to find the first one, and assess the difference between it and all the other leaf node heights. Since we can exit early and return False if an imbalance is detected, we should use a depth first search and count the height as we go.

Let's use a stack and while loop to do the depth first search, tracking the height as we go.

```
def is_balanced(root):
    leaf_height = 0
    stack = Stack()
    stack.push((root, 0))
    while not stack.is_empty():
        (current, height) = stack.pop()
        if current.left is None and current.right is None:
            if leaf_height != 0:
                if abs(height - leaf_height) > 1:
                    return False
            else:
                leaf_height = height
        if current.left is not None:
            stack.push((current.left, height + 1)
        if current.right is not None:
            stack.push(current.right, height + 1)
    return True
```

Notes based on book solution:

- The book finds the max depth recursively, then finds the min depth recursively, then computes the difference. This is less efficient than my solution in cases where the tree is unbalanced, and equal in efficiency in cases where the tree is balanced.

Problem 4.2: *Graph Connections*. Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Solution

We basically want to start at one of the nodes and try to get to the other node. We have the choice to use a depth first search or a breadth first search, but regardless of which approach we choose we have to check all the possible paths. We should also mark nodes as visited or not visited so that we don't end up in a cycle. Lastly, we should repeat the algorithm starting from

the other node and try to get to the first node, since the way in which the nodes are directed could prevent us from getting there.

My approach will be to copy the graph, use a depth first search via a stack and mark each node as visited. If I reach the target node return true, otherwise I add all unvisited connections to the stack, if the stack becomes empty I stop. I then delete the first copy and make a new one to reset all the visited flags. I repeat the algorithm starting from the target node and searching for the other node. This time, if the stack becomes empty, I return False.

```
def is_connected(graph_, node1, node2, first_time=True):
    graph = graph_.copy()
    stack = Stack()
    stack.push(node1)
    while not stack.is_empty():
        current = stack.pop()
        if current is node2:
            return True
        current.visited = True
        for connection in current.connections:
            if not hasattr(connection, “visited”):
                stack.push(connection)

    if first_time:
        return is_connected(graph_, node2, node1, first_time=False)
    else:
        return False
```

Notes based on book solution:

- The book doesn’t try the reverse order of the nodes, but I suppose that is something that can be left up to the user. Otherwise, the solutions are identical up to some Python quirks

Problem 4.3: *Plant a Tree*. Given a sorted (increasing order) array, write an algorithm to create a binary tree with minimal height.

Solution

We’ll treat our tree as a binary search tree. We need to make sure we’re always trying to insert the midpoint of the array to keep the tree balanced though.

```
class BSTNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def insert(self, root, data):
        if root is None:
```

```
            root = BSTNode(data)
        if data < root.data:
            self.insert(root.left)
        elif data > root.data:
            self.insert(root.right)

def make_balanced_tree(arr, tree=None):
    if len(arr) == 0:
        return

    midpoint = len(arr) // 2
    if tree is None:
        tree = BSTNode(arr[midpoint])
    else:
        tree.insert(arr[midpoint])

    make_balanced_tree(arr[:midpoint], tree)
    make_balanced_tree(arr[midpoint:], tree)
```

This should work, but a possible follow-up twist is how would you do it if the array were not sorted to begin with. Option A is to sort the array, since making the tree will already be O(n log(n)). Option B is to use a self-balancing tree such as an AVL tree.

Problem 4.4: *Depth Linked Lists*. Given a binary search tree, define an algorithm to create a linked list for each depth level.

Solution

This feels like all we need to do is a breadth first search and every time we start a new level, start a new linked list. How do we know when we've started a new level? If we track the first element in the level (starting with the root), once we hit an element that is smaller, we know we've started a new level.

Unfortunately, this only works if the tree is complete, or at minimum if the presence of a leftmost child is always guaranteed. Perhaps as we go through a level creating a linked list, we can scan for the first child (we'll already be doing this to add the children to the queue, so if we track the data of the first child then we'll know where to start the next linked list).

```
class LinkedListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def build_depth_lists(root):
    if root is None:
        return []
```

```
    # Find first child
    if root.left is not None:
        first_child = root.left
    elif root.right is not None:
        first_child = root.right
    else:
        return [LinkedListNode(root.data)]

    queue = Queue()
    output = []
    prev, head = None, None
    queue.enqueue(root)
    while not queue.is_empty():
        current = queue.dequeue()
        if current is first_child:
            output.append(head)
            head = first_child
            first_child = None
        else:
            node = LinkedListNode(current.data)
            if prev is not None:
                prev.next = node
            else:
                head = node
            prev = node

        if current.left is not None:
            if first_child is None:
                first_child = current.left
        queue.enqueue(current.left)

        if current.right is not None:
            if first_child is None:
                first_child = current.right
        queue.enqueue(current.right)

    return output
```

Notes based on book solution:

- None

Problem 4.5: *Inorder Successor*. Write an algorithm to find the ‘next’ node (e.g., in-order successor) of a given node in a binary search tree where each node has a link to its parent.

Solution

If the node is a left child, its inorder successor will be its parent. If the node is a right child, the will be the leftmost child in the subtree to the right of its parent node. Traversing the tree inorder

is easy, because you can do it recursively from the root. Starting an inorder traversal at an arbitrary node is slightly more difficult.

If the node is a right child, we need to continue up the tree of parents until a parent has a right child that is not the node we were previously at. Then we need to find the leftmost child of that node.

```
def find_inorder_successor(node):
    # Case 0: node is invalid or root
    if node is None or node.parent is None:
        return None

    # Case 1: node is a left child
    if node.parent.left is node:
        return parent

    # Case 2: node is a right child
    current = node
    while node.parent.right is current:
        current = current.parent
    while current.left is not None:
        current = current.left
    return current
```

Notes based on book solution:

- Case 2 could also be done recursively

Problem 4.6: *First Common Ancestor*. Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

Solution

Assuming the nodes have a connection to their parent, we can traverse the tree upward, marking the nodes along the way. If either node hits a marked node, we have the first common ancestor. The process terminates when both paths reach the root, in which case the root is the first common ancestor.

```
def find_first_common_ancestor(node1, node2):
    curr_1 = node1
    curr_2 = node2

    keep_going = curr_1.parent is not None or curr_2.parent is not None
    while keep_going:
        if curr_1.parent is not None:
            if hasattr(curr_1.parent, “visited”):
                return curr_1.parent
            else:
```

```
                curr_1.visited = None
                curr_1 = curr_1.parent

        if curr_2.parent is not None:
            if hasattr(curr_2.parent, “visited”):
                return curr_2.parent
            else:
                curr_2.visited = None
                curr_2 = curr_2.parent

    return curr_1  # both curr_1 and curr_2 are root
```

Notes based on book solution:

- None

Problem 4.7: *Large Trees*. You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide if T2 is a subtree of T1.

Solution

The key will be to search through the large tree as infrequently as possible. Brute force would be to search the large tree for the root of T2, and then to compare all the nodes in T2 with their suspected counterparts.

Unfortunately I can’t think of anything better than this approach, but neither can the book, so the approach is a breadth first search for the root of T2, and then start comparing nodes. If we find an inconsistency, continue the breadth first search for another instance of the root of T2 in T1.

Problem 4.8: *Print Paths*. You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum up to that value. Note that it can be any path in the tree - it does not have to start at the root.

Solution

This feels like a depth first search, where we continue adding depth if we are less than the sum, print the path when we equal the sum, and remove depth when we are larger than the sum. We have to figure out how to deal with paths that do not start at the root though.

Let’s move the start of the path through the tree in a BFS manner. Then for each path start, we can do a depth first search where we terminate the search if the sum exceeds the desired sum. Algorithmically, this is quite slow because nodes will be considered in several depth first searches as we try more start points. This approach will also be confused by negative numbers present in the tree.

If negative numbers can be in the tree, then the paths can be arbitrarily long. When adding each additional element, we could also ask can I get to the sum by removing from the start of the

path. This approach would involve log(n) previous checks for each step we take, so overall that would be O(n log(n)), which is not too bad. It also removes the need for a BFS since we're always checking different start points.

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 5: Bit Manipulation

Problem 5.1: *Bit Manipulation*. You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to set all bits between i and j in N equal to M (e.g., M becomes a substring of N located at i and starting at j).

Solution

We basically want to overwrite bits in N. We want to clear the bits of interest in N, select bits of interest in M, add the selected bits to N.

```
def overwrite_bits(N, M, i, j):
    assert M < N, “The larger number must be the first argument”
    assert i <= j, “The larger index must be the last argument”

    # Make a bitmask for the bits from i to j (inclusive)
       mask = abs(1 - (1 << (j-1)) << i)

    # Clear bits in N between index i and index j
    N &= ~mask

    # Replace the cleared bits with M
    N += (M << i) & mask

    return N
```

Problem 5.2: *Bit Manipulation*. Given a (decimal - e.g. 3.72) number that is passed in as a string, print the binary representation. If the number can not be represented accurately in binary, print “ERROR”

Solution

Decimals can be represented in binary as sums of powers of 0.5, like integers are represented as sums of powers of 2. Here we can iterate through the string and track which one-half bits are set. If the string is longer than 32 decimal places, it’s decimals will not be able to be represented in binary.

Problem 5.3: *Bit Manipulation*. Given an integer, print the next smallest and next largest number that have the same number of 1 bits in their binary representation.

Solution

So the numbers we have access to are constrained by the total number of set bits. We could start at our number, subtract 1, count the number of set bits, and continue until we find a

number with the same number of set bits. We could do the same process for addition, but we would add 1 each time. This feels somewhat inefficient.

If we could generate a list of all numbers with the allowed number of set bits, we could sort the list to find the next and previous allowed number, but this too feels inefficient.

Let's walk through an example. Say we start with 9 = 1001 which has two set bits. The results would be 10 = 1010 and 5 = 0101. To find 10, we can toggle the rightmost two bits, and two find 5 we can toggle the leftmost two bits. Let's see how that approach holds up for numbers where the leftmost or rightmost bits are the same like 8 = 1000 and 12 = 1100.

For 8, the result is 16 = 10000 and 4 = 0100. Here the rightmost two bits were the same, and we ended up toggling the largest two bits to find the greater number. To find the smaller number we toggle the most significant bit with the next most significant bit. For 12, the result is 24 = 11000 and 10 = 1010.

Writing out the different cases:

1. To find the larger number, we unset the LSB and set the next most significant unset bit
2. To find the smaller number, we unset the MSB and set the next most significant unset bit

```
import math

def find_next_largest(num):
    # Find the lsb
    lsb = num & -num
    lsb_index = int(math.log(lsb, 2))

    # Find the next most significant unset bit - lsb of shifted complement
    shifted_complement = ~num >> lsb_index
    next_most_significant = (shifted_complement & -shifted_complement) << lsb_index

    return num + next_most_significant - lsb

def find_next_smallest(num):
    # Find the msb
    msb_index = int(math.log(num, 2))

    # Find the next most significant unset bit - msb of complement
    next_most_significant_index = int(math.log(~num, 2))

    return num - 2**msb_index + 2**next_most_significant_index

def find_next_numbers(num):
    print("Next smallest: ", find_next_smallest(num))
    print("Next largest: ", find_next_largest(num))
```

Notes based on book solution:

- The book traverses the bits in the number and has 2 pages of code. Thumbs down.

Robert Morgan
Dec. 2020, Jan. 2021

Problem 5.4: *Bit Manipulation*. Explain what the following code does: ((n & (n-1)) == 0).

Solution

Starting inward and working outward, we are comparing a number with the number one below it. This bitwise AND comparison will be 1 if the number and the number one less than it have the bit set and 0 otherwise. Then we determine if the result is 0. The result will be zero if the number and the number one less than it have no common bits. The only time this is the case is when the number is a power of 2. Therefore, this expression determines if n is a power of 2.

Notes based on book solution:

- The expression will also evaluate to True for 0, so watch out for that

Problem 5.5: *Bit Manipulation*. Write a function to determine the number of bits required to convert integer A to integer B.

Solution

By convert, I'm going to assume this means how many bits have to be changed to convert one number to another. The example given converts 31 to 14 by unsetting the 4th bit and unsetting the 0th bit, so the result is two.

We need to count which bits are different between the two numbers, and to do that we use the XOR operator. Then we can count the number of set bits in the result of the XOR

```
def count_set_bits(num):
    count = 0
    while num > 0:
        num &= (num - 1)
        count += 1
    return count

def count_conversion(num1, num2):
    return count_set_bits(num1 ^ num2)
```

Notes based on book solution:

- None

Problem 5.6: *Bit Manipulation*. Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, etc).

Solution

Robert Morgan
Dec. 2020, Jan. 2021

To swap bits, I'm thinking we want to compare the number to a shifted version of itself. This way the 0 and 1 bit are compared, the 2 and 3 bit are compared. If we compare with the XOR operator, we'll know if they are the same or different. If they are the same, we don't need to worry about them, and if they are different, then we need to swap.

Let's walk through an example. Compare 1001010100001011 shifted to itself.

```
10010101000010110
^1001010100001011
------------------
11011111100011101
```

For the 0 -> 1 comparison, the result is 1, so we'll need to swap. The next bit in the result is the 1 -> 2 comparison, which we don't care about, so we skip it. We repeat this procedure throughout the rest of the XOR result, skipping every other bit.

To actually swap the bits (which we know are different), we need to unset the set bit and set the unset bit. We can iterate through the number, and in instances where we know we need to swap check if the bit is set, and this check will also tell us what we need to do to the other bit.

```
def swap_adjacent(num):
    xor_shift = (num << 1) ^ num
    mask = 1
    while mask < xor_shift:
        if xor_shift & mask:
            # need to swap detected
            if mask & num:
                # unset current bit and set next bit
                num = num - 2**mask + 2**(mask+1)
            else:
                # set current bit and unset next bit
                num = num + 2**mask - 2**(mask+1)

        mask = mask << 2
```

Notes based on book solution:

- Even faster: create a mask for all the odd bits and a mask for the even bits. Then select the even and odd bits separately, shift and combine

Problem 5.7: *Bit Manipulation*. An array A[1...n] contains all the integers from 0 to n except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of A[i]", which takes constant time. Write code to find the missing integer. Can you do it in O(n) time?

Solution

Robert Morgan
Dec. 2020, Jan. 2021

We can access a single bit of a single element of the array with the `fetch(arr, i, j)` function. Since the array is sorted, we know every other bit should have the zeroth bit set. We can go through the array inspecting the 0th bit of every element. Once we find a bit that is set when it's not supposed to be, or vice versa, we have found the missing number.

```
def find_missing_number(arr):
    # Find the index of the first discrepancy
    i = 0
    n = len(arr) + 1
    while i < n:
        if not (i % 2) & fetch(arr, i, j=0):
            break
        i += 1
    else:
        raise ValueError("No missing integers in array")

    # Now we construct the full integer using fetch
    out_int = 0
    j = 0
    while True:
        try:
            out_int += fetch(arr, i, j)
            j += 1
        except IndexError:
            break

    return out_int - 1
```

Notes based on book solution:

- None

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 8: Recursion

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 9: Sorting and Searching

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 12: System Design and Memory Limits