Udacity Course - Data Structure & Algorithms

Introduction

How to Solve Problems:

The Problem:

Given your birthday and the current date, calculate your age in days. Compensate for leap years. Assume that the birthday and current date are correct dates (no time travel). Simply put, if you were born 1 Jan 2012 and today's date is 2 Jan 2012 you are 1 day old.

- Dont Panic
- Possible inputs? What are the inputs?
    - Given your birthday and the current date, calculate your age in days.
    - Inputs : two dates
        - Checks inputs:
            - Second date must not be before first date
- What are the outputs?
    - Return a number giving the number of days between the first date and the second date
- Solving the problem
    - Work out examples - understand the relationship
        - Create examples, for each give the expected output or undefined if there is no defined output.
            - daysBetweenDates(2012, 12, 7, 2012, 12, 7) -> 0
            - daysBetweenDates(2012, 12, 7, 2012, 12, 8) -> 1
            - daysBetweenDates(2012, 12, 8, 2012, 12, 7) -> undefined
            - daysBetweenDates(2012, 6, 29, 2013, 6, 29) -> 365
            - daysBetweenDates(2012, 6, 29, 2013, 6, 31) -> undefined
        - Resolve examples by hand first
            - daysBetweenDates(2013,1,24,2013,6,29)

```
Jan 24 - Jan 31 = 7
+ Feb            28
+ March          31
+ April          30
+ May            31
+ Jun 1 - 29     29
Total            156
```

- Simple mechanical solution - simplify the solution
  - Pseudocode:

```
days = 0
while date1 is before date2:
    date1 = day after date1
    days += 1
return days
```

  - Break the problem in small peaces, and start by the order that you think is best.
    - First lets try to code nextDay(year, month,day)

```
def nextDay(year, month, day):
    """ Warning: this version incorrectly
    assumes all months have 30 days!"""
    if day < 30:
        return year, month, day + 1
    else:
        if month < 12:
            return year, month + 1, 1
        else:
            return year + 1, 1, 1
```

  - Test every peace of the code, to know your are in the right track
    - Example of tests:

```
def test():
    # tests with 30-day months!
    assert daysBetweenDates(2013, 1, 1, 2013, 1, 1) ==
0
    assert daysBetweenDates(2013, 1, 1, 2013, 1, 2) ==
1
    assert nextDay(2013, 1, 1) == (2013, 1, 2)
    assert nextDay(2013, 4, 30) == (2013, 5, 1)
    assert nextDay(2012, 12, 31) == (2013, 1, 1)
    print "Tests finished."
```

Summary

1. What are the inputs
2. What are the outputs

3. Work through some examples by hand
4. Simple mechanical solution
5. Develop incrementally and test as we go

Efficiency

Let's look again at this function from above:

```
def say_hello(n):
    for i in range(n):
        print("Hello!")
```

As the input increases, the number of lines executed also increases.

But we can go further than that! We can also say that as the input increases, the number of lines executed increases by a proportional amount. Increasing the input by 1 will cause 1 more line to get run. Increasing the input by 10 will cause 10 more lines to get run. Any change in the input is tied to a consistent, proportional change in the number of lines executed. This type of relationship is called a **linear relationship**.

Now here's a slightly modified version of the say_hello function:

```
def say_hello(n):
    for i in range(n):
        for i in range(n):
            print("Hello!")
```

Notice that when the input goes up by a certain amount, the number of operations goes up by the square of that amount. If the input is 2, the number of operations is 2^2. If the input is 3,the number of operations is 3^2 or 9.

To state this in general terms, if we have an input, n, then the number of operations will be n^2. This is what we would call a quadratic rate of increase.

We will use this graph as a referece and reminder of the importance of the run time of an algorithm. We have the number of inputs (n) on the X axis and the the number of operations required (N) on the Y axis.

Notice that if n is very small, it doesn't really matter which function we use—but as we put in larger values for n, the function with the nested loop will quickly become far less efficient.

Order

We should note that when people refer to the rate of increase of an algorithm, they will sometimes instead use the term order. Or to put that another way:

The rate of increase of an algorithm is also referred to as the order of the algorithm.

Big O Notation

Video explanation of how to get the big O of O(n)

When describing the efficiency of an algorithm, we could say something like "the run-time of the algorithm increases linearly with the input size". This can get wordy and it also lacks precision. So as an alternative, mathematicians developed a form of notation called big O notation. The "O" in the name refers to the order of the function or algorithm in question.

O(n):

```
def say_hello(n):
    for i in range(n):
        print("Hello!")
```

O(n^2):

```
def say_hello(n):
    for i in range(n):
        for i in range(n):
            print("Hello!")
```

We've been approximating efficiency by counting the number of lines of code that get executed. But when we are thinking about the run-time of a program, what we really care about is how fast the computer's processor is, and how many operations we're asking the processor to perform. Different lines of code may demand very different numbers of operations from the computer's processor. For now, counting lines will work OK as an approximation, but as we go through the course you'll see that there's a lot more going on under the surface.

In n^2 + 5n, the 5 has very little impact on the total efficiency—especially as the input size gets larger and larger. Asking the computer to do 10,005 operations vs. 10,000 operations makes little difference. Thus, it is the n^2 that we really care about the most, and the +5 makes little difference.

 big-O - https://www.bigocheatsheet.com/

Quadratic Example

```
# O(n^2)

def Quad_Example(our_list):
    for first_loop_item in our_list:
        for second_loop_item in our_list:
            print ("Items: {}, {}".format(first_loop_item,second_loop_item))


Quad_Example([1,2,3,4])
```

```
%time
```

```
Items: 1, 1
Items: 1, 2
Items: 1, 3
Items: 1, 4
Items: 2, 1
Items: 2, 2
Items: 2, 3
Items: 2, 4
Items: 3, 1
Items: 3, 2
Items: 3, 3
Items: 3, 4
Items: 4, 1
Items: 4, 2
Items: 4, 3
Items: 4, 4
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 6.2 µs
```

Log Linear Example

```
# O(nlogn)

# Don't worry about how this algorithm works, we will cover it later in the
course!

def Log_Linear_Example(our_list):

    if len(our_list) < 2:
        return our_list

    else:
        mid = len(our_list)//2
        left = our_list[:mid]
        right = our_list[mid:]

        Log_Linear_Example(left)
        Log_Linear_Example(right)

        i = 0
        j = 0
        k = 0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
```

```
                our_list[k]=left[i]
                i+=1
            else:
                our_list[k]=right[j]
                j+=1
            k+=1

        while i < len(left):
            our_list[k]=left[i]
            i+=1
            k+=1

        while j < len(right):
            our_list[k]=right[j]
            j+=1
            k+=1

        return our_list

Log_Linear_Example([56,23,11,90,65,4,35,65,84,12,4,0])

%time
```

Linear Example

```
# O(n)

def Linear_Example(our_list):
    for item in our_list:
        print ("Item: {}".format(item))

Linear_Example([1,2,3,4])

%time
```

Logarithmic Example

```
# O(logn)

def Logarithmic_Example(number):
    if number == 0:
        return 0

    elif number == 1:
        return 1

    else:
        return Logarithmic_Example(number-1)+Logarithmic_Example(number-2)
```

```
Logarithmic_Example(29)

%time
```

Constant Example

```
# O(1)

def Constant_Example(our_list):
    return our_list.pop()

Constant_Example([1,2,3,4])

%time
```

Space Complexity Examples

When we refer to space complexity, we are talking about how efficient our algorithm is in terms of memory usage. This comes down to the datatypes of the variables we are using and their allocated space requirements. In Python, it's less clear how to do this due to the underlying data structures using more memory for house keeping functions (as the language is actually written in C).

For example of the learning experience:

Type	Storage size
char	1 byte
bool	1 byte
int	4 bytes
float	4 bytes
double	8 bytes

Example 1

```
def our_constant_function():

    x = 3 # Type int
    y = 345 # Type int
    z = 11 # Type int

    answer = x+y+z

    return answer
```

So in this example we have four integers (x, y, z and answer) and therefore our space complexity will be 4*4 = 16 bytes. This is an example of constant space complexity, since the amount of space used does not change with input size.

Example 2

```
def our_linear_function(n):

    n = n # Type int
    counter = 0 # Type int
    list_ = [] # Assume that the list is empty (i.e., ignore the fact that there
is actually meta data stored with Python lists)

    while counter < n:
        list_.append(counter)
        counter = counter + 1

    return list_
```

So in this example we have two integers (n and counter) and an expanding list, and therefore our space complexity will be 4*n + 8 since we have an expanding integer list and two integer data types. This is an example of linear space complexity.

[https://courses.cs.northwestern.edu/311/html/space-complexity.html]

```
    int sum(int x, int y, int z) {
    int r = x + y + z;
    return r;
    }
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

```
int sum(int a[], int n) {
int r = 0;
for (int i = 0; i < n; ++i) {
    r += a[i];
}
return r;
}
```

requires N units for a, plus space for n, r and i, so it's O(N). What are the space complexities of these next two functions?

```
void matrixAdd(int a[], int b[], int c[], int n) {
for (int i = 0; i < n; ++i) {
    c[i] = a[i] + b[j]
}
}
void matrixMultiply(int a[], int b[], int c[][], int n) { // not legal C++
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
    c[i] = a[i] + b[j];
    }
}
}
```

If a function A uses M units of its own space (local variables and parameters), and it calls a function B that needs N units of local space, then A overall needs M + N units of temporary workspace. What if A calls B 3 times? When a function finishes, its space can be reused, so if A calls B 3 times, it still only needs M + N units of workspace.

What if A calls itself recursively N times? Then its space can't be reused because every call is still in progress, so it needs O(N2) units of workspace.

But be careful here. If things are passed by pointer or reference, then space is shared. If A passes a C-style array to B, there is no new space allocated. If A passes a C++ object by reference to B, there is no new space. What if A passes a vector or string by value? Most likely, new space will be allocated. Some C++ compilers try to avoid this for strings, using a technique called copy-on-write, but this is no longer a common thing to do.

Explanation of how to calculate space complexity here.

Data Structures

Array and Linked Lists

Collections

Properties of colletions:

- Don't have a particular order (so you can't say "give me the third element in this collection")
- Don't have to have objects of the same type.

Data structures are extensions of colletions that get there properties and sum new ones

List

A list has all the properties of a collections but the objects have a order

- Have an order (so you can say things like "give me the 3rd item in the list")
- Have no fixed length (you can add or remove elements)

Arrays vs lists vs list

One of the key differences is that arrays have indices, while lists do not.

An array is a place in memory continuous that has the same size. And that's why it have index, since we know the location in memory of one element we know that the other's are side by side.

In constrast the elements of a list can be or not next to one another in memory.

> A Python list is essentially implemented like an array (specifically, it behaves like a dynamic array, if you're curious). In particular, the elements of a Python list are contiguous in memory, and they can be accessed using an index.

And the Python list has more functionality that an array like the methods pop() and append()

Strings

Strings Methods

Python Strings are Arrays. They are arrays of bytes representing unicode characters (char *str - in language C)

Linked Lists

Made of nodes that has a value and a pointer (reference)

Create a Linked List:

```
# Creating a Node class with value and next attributes
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

## Create the head of the linked list and give value 2
head = Node(2)
# Create another Node with value 1 and linked head to that Node
head.next = Node(1)

# To print the value o the next Node:
print(head.next.value)
```

Create a linked list using iteration

```
def create_linked_list_better(input_list):

    head = None
    tail = None
```

```
    for value in input_list:

        if head is None:
            head = Node(value)
            tail = head
        else:
            tail.next = Node(value)
            tail = tail.next        # update the tail

    return head
```

Traversing the list

One way to pass through the linked list and print is values is:

```
def print_linked_list(head):
    # Declare the current node has the beginning of the list
    current_node = head

    # While is not the end of the linked list
    while current_node is not None:
        print(current_node.value)
        # Directionate the current node to the next in the linked list
        current_node = current_node.next

print_linked_list(head)
```

Singly Linked Lists

Usually you'll want to create a LinkedList class as a wrapper for the nodes themselves and to provide common methods that operate on the list. For example you can implement an append method that adds a value to the end of the list.

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Method to append in a linked list

```
    # If the linkedList is in the beggining
    def append(self, value):
        if self.head is None:
            self.head = Node(value)
```

```
            return
        # if the linked list already has more nodes

        # Move to the tail (the last node)
        node = self.head
         # while node next has values to another node continue
        while node.next:
            node = node.next

        # reached the end of the linked list
        node.next = Node(value)
        return
```

Method to transforms a linked list in a list

```
    def to_list(self):
        out_list = []

        node = self.head
        while node:
            out_list.append(node.value)
            node = node.next

        return out_list
```

Method Prepend a node to the beginning of the list

```
    def prepend(self, value):
        """ Prepend a node to the beginning of the list """

        if self.head is None:
            self.head = Node(value)
            return

        new_head = Node(value)
        new_head.next = self.head
        self.head = new_head
```

Serch() function that search by value

```
    def search(self, value):
        """ Search the linked list for a node with the requested value and return
the node. """
```

```
        if self.head is None:
            return None

        node = self.head
        while node:
            if node.value == value:
                return node
            node = node.next

        raise ValueError("Value not found in the list.")
```

Remove() remove a nove

Pseudocode

Linked List:

1. Create two pointers (pointer 1 points to the head, points 2 point to the head next)
2. Advnce until point2 find the target
3. Create another pointer (pointer3) that points to the node that point2 is poiting
4. Advance point2 to target next
5. pointer1 to point to node that pointer2 is aiming
6. Remove pointer 3

```
def remove(self, value):
    """ Delete the first node with the desired data. """
    if self.head is None:
        return

    if self.head.value == value:
        self.head = self.head.next
        return

    node = self.head
    while node.next:
        if node.next.value == value:
            node.next = node.next.next
            return
        node = node.next

    raise ValueError("Value not found in the list.")
```

pop() functions that removes the last element in the linked list

```
    def pop(self):
    """ Return the first node's value and remove it from the list. """
    if self.head is None:
        return None

    node = self.head
    self.head = self.head.next

    return node.value
```

insert()

Pseudocode:

Linked List:

1. Create a new pointer that points to the head
2. Seek up two but not including to the node that we want to
3. Create next node
4. Make new node point to the node target
5. Previous node points to new node

```
  def insert(self, value, pos):
      """ Insert value at pos position in the list. If pos is larger than the
          length of the list, append to the end of the list. """
      # If the list is empty
      if self.head is None:
          self.head = Node(value)
          return

      if pos == 0:
          self.prepend(value)
          return

      index = 0
      node = self.head
      while node.next and index <= pos:
          if (pos - 1) == index:
              new_node = Node(value)
              new_node.next = node.next
              node.next = new_node
              return

          index += 1
```

```
            node = node.next
        else:
            self.append(value)
```

size() functions and test its functionality

```
    def size(self):
        """ Return the size or length of the linked list. """
        size = 0
        node = self.head
        while node:
            size += 1
            node = node.next

        return size
```

Reversing a linked list exercise

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

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

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
            return

        node = self.head
        while node.next:
            node = node.next

        node.next = Node(value)

    def __iter__(self):
        node = self.head
        while node:
            yield node.value
            node = node.next

    def __repr__(self):
        return str([v for v in self])
```

```
def reverse(linked_list):
    """
    Reverse the inputted linked list

    Args:
    linked_list(obj): Linked List to be reversed
    Returns:
    obj: Reveresed Linked List
    """
    new_list = LinkedList()

    prev_node = None

    """
    A simple idea - Pick a node from the original linked list traversing form the
beginning, and
    prepend it to the new linked list.
    We have to use a loop to iterate over the nodes of original linked list
    """
    # In this "for" loop, the "value" is just a variable whose value will be
updated in each iteration
    for value in linked_list:
        # create a new node
        new_node = Node(value)

        # Make the new_node.next point to the
        # node created in previous iteration
        new_node.next = prev_node

        # This is the last statement of the loop
        # Mark the current new node as the "prev_node" for next iteration
        prev_node = new_node

    # Update the new_list.head to point to the final node that came out of the
loop
    new_list.head = prev_node

    return new_list
```

Doubly Linked Lists

This type of list has connections backwards and forwards through the list.

```
class DoubleNode:
    def __init__(self, value):
        self.value = value
        self.next = None
        self.previous = None
```

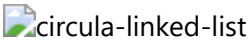
```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    # Implementing the method append to  double linked list
    def append(self, value):
        if self.head is None:
            self.head = DoubleNode(value)
            self.tail = self.head
            return

        self.tail.next = DoubleNode(value)
        self.tail.next.previous = self.tail
        self.tail = self.tail.next
        return
```

Circular Linked Lists

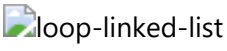
Circular linked lists occur when the chain of nodes links back to itself somewhere. For example `NodeA -> NodeB -> NodeC -> NodeD -> NodeB` is a circular list because `NodeD` points back to `NodeB` creating a loop `NodeB -> NodeC -> NodeD -> NodeB`.

 circula-linked-list

Detecting Loops in Linked

In this notebook, you'll implement a function that detects if a loop exists in a linked list. The way we'll do this is by having two pointers, called "runners", moving through the list at different rates. Typically we have a "slow" runner which moves at one node per step and a "fast" runner that moves at two nodes per step.

If a loop exists in the list, the fast runner will eventually move behind the slow runner as it moves to the beginning of the loop. Eventually it will catch up to the slow runner and both runners will be pointing to the same node at the same time. If this happens then you know there is a loop in the linked list. Below is an example where we have a slow runner (the green arrow) and a fast runner (the red arrow).

 loop-linked-list

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

class LinkedList:
    def __init__(self, init_list=None):
        self.head = None
        if init_list:
            for value in init_list:
                self.append(value)
```

```
    def append(self, value):
        if self.head is None:
            self.head = Node(value)
            return

        # Move to the tail (the last node)
        node = self.head
        while node.next:
            node = node.next

        node.next = Node(value)
        return

list_with_loop = LinkedList([2, -1, 3, 0, 5])

# Creating a loop where the last node points back to the second node
loop_start = list_with_loop.head.next

node = list_with_loop.head
while node.next:
    node = node.next
node.next = loop_start

# Solution

def iscircular(linked_list):
    """
    Determine wether the Linked List is circular or not

    Args:
    linked_list(obj): Linked List to be checked
    Returns:
    bool: Return True if the linked list is circular, return False otherwise
    """

    if linked_list.head is None:
        return False

    slow = linked_list.head
    fast = linked_list.head

    while fast and fast.next:
        # slow pointer moves one node
        slow = slow.next
        # fast pointer moves two nodes
        fast = fast.next.next

        if slow == fast:
            return True
```

```
    # If we get to a node where fast doesn't have a next node or doesn't exist
itself,
    # the list has an end and isn't circular
    return False
```

Stacks

[https://runestone.academy/runestone/books/published/pythonds/BasicDS/WhatisaStack.html]

For example, every web browser has a Back button. As you navigate from web page to web page, those pages are placed on a stack (actually it is the URLs that are going on the stack). The current page that you are viewing is on the top and the first page you looked at is at the base. If you click on the Back button, you begin to move in reverse order through the pages.

The stack operations are given below.

- Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.
- push(item) adds a new item to the top of the stack. It needs the item and returns nothing.
- pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items on the stack. It needs no parameters and returns an integer.
- Order **O(1)**
- L.I.F.O - Last In, First Out

Queues

[https://runestone.academy/runestone/books/published/pythonds/BasicDS/WhatIsaQueue.html]

Computer science also has common examples of queues. Our computer laboratory has 30 computers networked with a single printer. When students want to print, their print tasks “get in line” with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you

As an ordered collection of items which are added at one end, called the “rear,” and removed from the other end, called the “front.”

- FIFO - First In, First Out
- Head - the first element of the queue (the oldest)
- Tail - the last element of the queue (the youngest)
- Enqueue - add a element to the tail
- Dequeue - remove the head element
- Peek - look at the head element but dont remove it

- Deques or Double ended queue - is a queue that goes both ways (head and tail)
- Priority Queue - each element has a priority number, when dequeue remove the element with the highest priority number
- Complexity of a linked list queue is **O(1)** (enqueue and dequeue)
- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer.

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- addFront(item) adds a new item to the front of the deque. It needs the item and returns nothing.
- addRear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- removeFront() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- removeRear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- isEmpty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

Adding and removing items from the front is O(1) whereas adding and removing from the rear is O(n).

Recursion

Video explaination

With recursion, we solve a problem by first solving smaller instances of the same problem. In practice, this often involves calling a function from within itself—in other words, we feed some input into the function, and the function produces some output—which we then feed back into the same function. And we continue to do this until we arrive at the solution.

So when should we use recursion (when both of this conditions happen)?

- When the problem has a tree-like structure
- When the problem requires backtracking

https://runestone.academy/runestone/books/published/pythonds/Recursion/WhatIsRecursion.html

- All recursive algorithms must obey three important laws:
  - A recursive algorithm must have a base case.
  - A recursive algorithm must change its state and move toward the base case.
  - A recursive algorithm must call itself, recursively.

Practice Problem

Implement sum_integers(n) to calculate the sum of all integers from 1 to n using recursion. For example, sum_integers(3) should return 6 (1+2+3).

```
def sum_integers(n):
    # Its called base case
    if n == 1:
        return 1

    # Its called recursive case
    return n + sum_integers(n - 1)

print(sum_integers(3))


def factorial(n):
    """
    Calculate n!

    Args:
    n(int): factorial to be computed
    Returns:
    n!
    """

    if n == 0:
        return 1  # by definition of 0!
    return n * factorial(n-1)

print ("Pass" if (1 == factorial(0)) else "Fail")
print ("Pass" if  (1 == factorial(1)) else "Fail")
print ("Pass" if  (120 == factorial(5)) else "Fail")
```

def is_palindrome(input): """ Return True if input is palindrome, False otherwise.

```
Args:
   input(str): input to be checked if it is palindrome
"""

# Termination / Base condition
if len(input) <= 1:
    return True
```

```
else:
    first_char = input[0]
    last_char = input[-1]

    # sub_input is input with first and last char removed
    sub_input = input[1:-1]

    # recursive call, if first and last char are identical, else return False
    return (first_char == last_char) and is_palindrome(sub_input)
```

print ("Pass" if (is_palindrome("")) else "Fail") print ("Pass" if (is_palindrome("a")) else "Fail") print ("Pass" if (is_palindrome("madam")) else "Fail") print ("Pass" if (is_palindrome("abba")) else "Fail") print ("Pass" if not (is_palindrome("Udacity")) else "Fail")

Trees

A tree has the following properties:

- One node of the tree is designated as the root node.
- Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.
- A unique path traverses from the root to each node.
- If each node in the tree has a maximum of two children, we say that the tree is a binary tree.
- The first element is called root
- It works like a linked list, but instead of pointing to one node, it can point to more

Tree Traversal: Process of visiting (checking and/or updating) each node in a tree data structure, exactly once.

There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.

Pre-order Traversal

- Check if the current node is empty / null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.


DPF tree

Pre-order: F, B, A, D, C, E, G, I, H.

In-Order Traversal

- Check if the current node is empty / null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.

First we go down as we can to the left and then we start display data of the current node, then by recursion do it again

In-order: A, B, C, D, E, F, G, H, I.

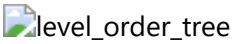
Post-Order Traversal

- Check if the current node is empty / null.
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of the root (or current node).

Display All left's, when no more left print right until more lefts

Post-order: A, C, E, D, B, H, I, G, F.

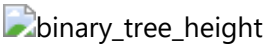
BFS - Breadth First Search (Level-Order Traversal)



BFS - Reverse Level-Order Traversal

Height of Binary Tree

- The height of a tree is the height of its root node.
- Height of Node:
    - The height of a node is the number of edges on the longest path between that node and a leaf.

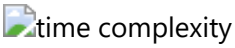


Size of Binary Tree

The total number of nodes in the tree

Binary Search Tree

Binary search trees differ from binary trees in that the entries are ordered.



BST Property

The BST property—every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller than the current node

The binary search tree property (BST property) is a global property that every binary search tree must satisfy.

 example property

Diameter of a Binary Tree

Diameter of a Binary Tree is the maximum distance between any two nodes

Maps and Hashing

- Maps == Dictionarys

```
udacity = {}
udacity['u'] = 1
```

Set

- No order
- No repeat values

```
Map = <key, value>
```

A group of keys is a set.

Hash Functions

Simply put, hash functions are these really incredible magic functions which can map data of any size to a fixed size data. This fixed sized data is often called hash code or hash digest

```
def hash_function(string):
    hash_code = 0
    for character in string:
        hash_code += ord(character)
    return hash_code

hash_code_1 = hash_function("abcd")
print(hash_code_1)
```

While doing the get() operation, if the entry is found in the cache, it is known as a cache hit. If, however, the entry is not found, it is known as a cache miss.

When two different inputs produce the same output, then we have something called a collision. An ideal hash function must be immune from producing collisions.

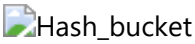
Similarly, we can treat abcde in base p as

$$a*p4+b*p3+c*p2+d*p1+e*p0$$

Here, we replace each character with its corresponding ASCII value.

A lot of research goes into figuring out good hash functions and this hash function is one of the most popular functions used for strings. We use prime numbers because the provide a good distribution. The most common prime numbers used for this function are 31 and 37.

Thus, using this algorithm, we can get a corresponding integer value for each string key and use it as an index of an array, say bucket array. It is not a special array. We simply choose to give a special name to arrays for this purpose. Each entry in this bucket array is called a bucket and the index in which we store a bucket is called bucket index. You can visualize the bucket array as shown in the figure below:



Compression Function

We now have a good hash function which will return unique values for unique objects. But let's look at the values. These are huge. We cannot create such large arrays. So we use another function - compression function to compress these values so as to create arrays of reasonable sizes.

A very simple, good, and effective compression function can be mod len(array). The modulo operator % returns the remainder of one number when divided by other.

So, if we have an array of size 10, we can be sure that modulo of any number with 10 will be less than 10, allowing it to fit into our bucket array.

Collision Handling

As discussed earlier, when two different inputs produce the same output, then we have a collision. Our implementation of get_hash_code() function is satisfactory. However, because we are using compression function, we are prone to collisions. Remember, that a key will always be unique. But the bucket_index generated by two different keys can be the same.

Consider the following scenario - We have a bucket array of length 10 and we get two different hash codes for two different inputs, say 355, and 1095. Even though the hash codes are different in this case, the bucket index will be same because of the way we have implemented our compression function. Such scenarios where multiple entries want to go to the same bucket are very common. So, we introduce some logic to handle collisions.

There are two popular ways in which we handle collisions.

- Separate chaining - Separate chaining is a clever technique where we use the same bucket to store multiple objects. The bucket in this case will store a linked list of key-value pairs. Every bucket has it's own separate chain of linked list nodes.
- Open Addressing - In open addressing, we do the following:
    - If, after getting the bucket index, the bucket is empty, we store the object in that particular bucket

- If the bucket is not empty, we find an alternate bucket index by using another function which modifies the current hash code to give a new code. This process of finding an alternate bucket index is called probing. A few probing techniques are - linear probing, qudratic probing, or double hashing.

Note: time complexity is always determined in terms of input size and not the actual amount of work that is being done independent of input size. That "independent amount of work" will be constant for every input size so we disregard that.

Now, the entire time complexity essentialy depends on the linked list traversal. In the worst case, all entries would go to the same bucket index and our linked list at that index would be huge. Therefore, the time complexity in that scenario would be $O(n)$. However, hash functions are wisely chosen so that this does not happen. On average, the distribution of entries is such that if we have n entries and b buckets, then each bucket does not have more than n/b key-value pair entries.

Therefore, because of our choice of hash functions, we can assume that the time complexity is $O(nb)$. This number which determines the load on our bucket array n/b is known as load factor.

Generally, we try to keep our load factor around or less than 0.7. This essentially means that if we have a bucket array of size 10, then the number of key-value pair entries will not be more than 7.

What happens when we get more entries and the value of our load factor crosses 0.7?

In that scenario, we must increase the size of our bucket array. Also, we must recalculate the bucket index for each entry in the hash map.

Note: the hash code for each key present in the bucket array would still be the same. However, because of the compression function, the bucket index will change.

Therefore, we need to rehash all the entries in our hash map. This is known as Rehashing.

Note: Theoretically, the worst case time complexity of `put` and `get` operations of a HashMap can be $O(\dfrac{n}{b}) \approx O(n)$, when $b < < n$. However, our hashing functions are sophisticated enough that in real-life we easily avoid collisions and never hit `O(n)`. Rather, for the most part, we can safely assume that the time complexity of `put` and `get` operations will be `O(1)`.

Therefore, when you are asked to solve any practice problem involving HashMaps, assume the worst case time complexity for `put` and `get` operations to be `O(1)`.

What is Caching?

Caching can be defined as the process of storing data into a temporary data storage to avoid recomputation or to avoid reading the data from a relatively slower part of memory again and again. Thus cachig serves as a fast "look-up" storage allowing programs to execute faster.

Huffman Coding

A data compression algorithm could be either lossy or lossless, meaning that when compressing the data, there is a loss (lossy) or no loss (lossless) of information. **The Huffman Coding** is a lossless data compression algorithm.

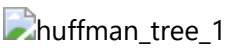
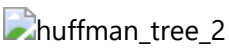
- Assume that we have a string message AAAAAAABBBCCCCCCCDDEEEEEE comprising of 25 characters to be encoded.

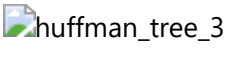
Phase I - Build the Huffman Tree

- First, determine the frequency of each character in the message. In our example, the following table presents the frequency of each character.

(Unique) Character	Frequency
A	7
B	3
C	7
D	2
E	6

- Second, each row in the table above can be represented as a node having a character, frequency, left child, and right child. In the next step, we will repeatedly require to pop-out the node having the lowest frequency. Therefore, build and sort a list of nodes in the order lowest to highest frequencies. Remember that a list preserves the order of elements in which they are appended.
- Pop-out two nodes with the minimum frequency from the priority queue created in the above step.
- Create a new node with a frequency equal to the sum of the two nodes picked in the above step. This new node would become an internal node in the Huffman tree, and the two nodes would become the children. The lower frequency node becomes a left child, and the higher frequency node becomes the right child. Reinsert the newly created node back into the priority queue.
- Do you think that this reinsertion requires the sorting of priority queue again? If yes, then a min-heap could be a better choice due to the lower complexity of sorting the elements, every time there is an insertion.
- Repeat steps #3 and #4 until there is a single element left in the priority queue. The snapshots below present the building of a Huffman tree.

- For each node, in the Huffman tree, assign a bit 0 for left child and a 1 for right child. See the final Huffman tree for our example: 

Phase II - Generate the Encoded Data

- Based on the Huffman tree, generate unique binary code for each character of our string message. For this purpose, you'd have to traverse the path from root to the leaf node.

(Unique) Character	Frequency	Huffman Code
D	2	000
B	3	001

(Unique) Character	Frequency	Huffman Code
E	6	01
A	7	10
C	7	11

This way, AAAAAAABBBCCCCCCCDDEEEEEE, our encoded data would be 101010101010100010010011111111111111100000010101010101

Source: Visualization of huffman codding

Algorithms

Complexity

To find out the efficiency of an algorithm you can make table of results that you can see the pattern and by doing a lot you can know what tipe is. table-binary Video explaining

calcullus-binary-complexity

Linear search

- What would the time complexity be for linear search?

  Order of n - **O(n)**

Binary search

Note that the word binary means "having two parts". Binary search means we are doing a search where, at each step, we divide the input into two parts. Also note that the data we are searching through has to be sorted.

- What would the time complexity be for linear search?

  Order of n - **O(log(n))**

In summary:

- Binary search is a search algorithm where we find the position of a target value by comparing the middle value with this target value.
- If the middle value is equal to the target value, then we have our solution (we have found the position of our target value).
- If the target value comes before the middle value, we look for the target value in the left half.
- Otherwise, we look for the target value in the right half.
- We repeat this process as many times as needed, until we find the target value

Example of implementing bynary code

```
def binary_search(array, target):
    start_index = 0
    end_index = len(array) - 1

    while start_index <= end_index:
        mid_index = (start_index + end_index)//2        # integer division in
Python 3

        mid_element = array[mid_index]

        if target == mid_element:                       # we have found the
element
            return mid_index

        elif target < mid_element:                      # the target is less than
mid element
            end_index = mid_index - 1                   # we will only search in
the left half

        else:                                           # the target is greater
than mid element
            start_index = mid_element + 1               # we will search only in
the right half

    return -1
```

Using a recursive way:

```
def binary_search_recursive_soln(array, target, start_index, end_index):
    if start_index > end_index:
        return -1

    mid_index = (start_index + end_index)//2
    mid_element = array[mid_index]

    if mid_element == target:
        return mid_index
    elif target < mid_element:
        return binary_search_recursive_soln(array, target, start_index, mid_index
- 1)
    else:
        return binary_search_recursive_soln(array, target, mid_index + 1,
end_index)
```

Example of contains.

The second variation is a function that returns a boolean value indicating whether an element is present, but with no information about the location of that element.

One option is just to wrap binary search:

```
def contains(target, source):
    return recursive_binary_search(target, source) is not None
```

Another choice is to build a simpler binary search directly into the function:

```
def contains(target, source):
    # Since we don't need to keep track of the index, we can remove the `left`
parameter.
    if len(source) == 0:
        return False
    center = (len(source)-1) // 2
    if source[center] == target:
        return True
    elif source[center] < target:
        return contains(target, source[center+1:])
    else:
        return contains(target, source[:center])
```

Trie

You've learned about Trees and Binary Search Trees. In this notebook, you'll learn about a new type of Tree called Trie. Before we dive into the details, let's talk about the kind of problem Trie can help with.

Let's say you want to build software that provides spell check. This software will only say if the word is valid or not. It doesn't give suggested words. From the knowledge you've already learned, how would you build this?

The simplest solution is to have a hashmap of all known words. It would take O(1) to see if a word exists, but the memory size would be O(n*m), where n is the number of words and m is the length of the word. Let's see how a Trie can help decrease the memory usage while sacrificing a little on performance.

Basic Trie Let's look at a basic Trie with the following words: "a", "add", and "hi"

```
basic_trie = {
    # a and add word
    'a': {
        'd': {
            'd': {'word_end': True},
            'word_end': False},
        'word_end': True},
    # hi word
    'h': {
```

```
         'i': {'word_end': True},
         'word_end': False}}


print('Is "a"   a word: {}'.format(basic_trie['a']['word_end']))
print('Is "ad"  a word: {}'.format(basic_trie['a']['d']['word_end']))
print('Is "add" a word: {}'.format(basic_trie['a']['d']['d']['word_end']))
```

You can lookup a word by checking if word_end is True after traversing all the characters in the word. Let's look at the word "hi". The first letter is "h", so you would call basic_trie['h']. The second letter is "i", so you would call basic_trie['h']['i']. Since there's no more letters left, you would see if this is a valid word by getting the value of word_end. Now you have basic_trie['h']['i']['word_end'] with True or False if the word exists.

In basic_trie, words "a" and "add" overlapp. This is where a Trie saves memory. Instead of having "a" and "add" in different cells, their characters treated like nodes in a tree. Let's see how we would check if a word exists in basic_trie.

```
def is_word(word):
    """
    Look for the word in `basic_trie`
    """
    current_node = basic_trie

    for char in word:
        if char not in current_node:
            return False

        current_node = current_node[char]

    return current_node['word_end']

# Test words
test_words = ['ap', 'add']
for word in test_words:
    if is_word(word):
        print('"{}" is a word.'.format(word))
    else:
        print('"{}" is not a word.'.format(word))
```

Trie Using a Class

Just like most tree data structures, let's use classes to build the Trie. Implement two functions for the Trie class below. Implement add to add a word to the Trie. Implement exists to return True if the word exist in the trie and False if the word doesn't exist in the trie.

```
class TrieNode(object):
    def __init__(self):
        self.is_word = False
        self.children = {}


class Trie(object):
    def __init__(self):
        self.root = TrieNode()

    def add(self, word):
        """
        Add `word` to trie
        """
        current_node = self.root

        for char in word:
            if char not in current_node.children:
                current_node.children[char] = TrieNode()
            current_node = current_node.children[char]

        current_node.is_word = True

    def exists(self, word):
        """
        Check if word exists in trie
        """
        current_node = self.root

        for char in word:
            if char not in current_node.children:
                return False
            current_node = current_node.children[char]

        return current_node.is_word
```

Heaps

Its is a specific type of tree

Rules:

- The elements are rearranged in decrease or increase order, since the root element is the max value or min of the tree
- There are two types of heaps, **max heaps and min heaps**, in a max heaps the parent is always bigger then the child, and in min is the opposite
- Heapify - is the operation that we reorganized the tree based on the heap property.

If it is a max heap, inserting a new value to the heap we have to find a open space insert the value, then rearranged that the parent has a bigger value

- Implementation - normaly they are stored in the arrays