Algorithms and Data Structure in Python

Neba Nfonsang
University of Denver

Introduction

- Algorithms are methods for solving a problem while data structure are methods of storing data.
- Effeciency is an important concept in algorithms and data structure. You want your code to be effecient in terms of runtime and memory space. You need to be able to get the job done with minimal resources (time and space).
- You can use algorithms to improve your code in terms of speed and space. Sometimes you will need a trade off between time and memory.

The Big O Notation

- Big O describes the effeciency of your code. It is a measure of the time complexity or runtime of an agorithm. It measures time effeciency as a function of the input. That is, if the input grows, how does the runtime grow? The time complexity could be modeled by a constant, linear, log, quadratic, exponential functions, etc.
- Time complexity or runtime analysis is important when you are dealing with a large input or dataset. To simplify the analysis, only operations with the highest degree of polynomial in an algorithm are considered since they impact the speed of the algorithm the most as input size increases.
- In analyzing the runtime of an algorithm, we can find the runtime or big O of every operation, then select the operation with the highest big O to be the big O of the algorithm.
- O(1) is constant runtime: This implies the runtime is independent of the input. An example could be poping from a stack or pushing and item to a stack or removing and appending an item to a list.
- O(n) is linear runtime: This implies the runtime increases linearly with the input. For exmaple, linear search algorithms follow a linear time complexity.
- O(n^2) is quadratic runtime: That means the algorithm's runtime increases as a quadratic function of the input. An example can be seen in an algorithm that has a nested for loop. For example, an algorithm that find all the possible pairs of items in a list.
- Big O analyzes worse case effeciencies

Binary Search

- A binary search algorithm searches a term in a sorted collection by comparing the search term with the middle term in the collection, then uses the the lower or upper half of the collection as a new search space based on results of the comparison. The search process is repeated in the new search space. The search space keeps reducing until the search term is found or not found.

Recursion

- Recursion is a method of solving complex problems by breaking the problem into subproblems.
- Recursion is useful for divide and conquer problems.
- In code, a recursive function is a function that calls itself within the function definition.
- Recursive algorithms could be more compact, elegant in code, and easier to understand.
- Recursive programs have a base case with a condition statement that indicates when the recursion should stop and a recursive case that calls the function itself within the function definition. A recursive call adresses subproblems but should not address subproblems that overlap.

Bubble Sort or Sinking sort

- This is a naive approach to sorting. The algorithm goes through the array and compares nearby elements and switch them such that the first element comes first. This process is repeated until all the elements are correctly sorted. The larger elements "bubbles to the top or towards the end of the collection" as the sorting process goes on.
- Each iteration through the list takes n-1 operations. For a worse case scenario where the first element was the largest, it would take n-1 iterations. So the time complexity would be f(n) = (n-1) x (n-1), resulting to O (n^2). However, the space complexity would be O(1) because no extra data structure was needed since the sorting was inplace.

Merge Sort

- This is a sorting algorithm that sorts a collection of items using a divide and conquer approach. That is, split the collection into smaller parts. Nearby parts are merged while sorting continues untill all the parts are merged into a whole collection that is completely sorted.

- The sort effeciency is approximately number of comparisons per step multiplied by number of steps, resulting to O(nlogn), which is better than that of bubble sort. The space efficiency is however, O(n) because we needed new arrays or data structure to copy our data into for every step.

Quick Sort

- Quick sort in most cases is the most effecient sorting algorithm. Quick sort picks one value as pivot and move all values larger than it to it's right and values lower than it to it's left. The values to the left and right are then compared to each other to check if they are sorted. If any elements on the left or the right of the pivot are not sorted, then, another pivot is selected and the process repeated in a new search space where the previous pivot acts like a lower or upper bound.
- The worse case scenario is when we already have the items sorted and our pivots starts with the last elment, then proceeds towards the first element. There would be n-1 steps with comparison from n-1 to n-2 etc such that the time complexity would be O(n^2). Quick sort is inplace, so the space complexity is constant, O(n).

The Shortest Path Problems

- The shortest path problem is all about finding the shortest path in a graph. Sometimes the edges can be weighted (with numerical values) or and sometimes we can have unweighted edges. the shortest path is the path with the smallest sum of the edges. With unweighted edges, the shortest path is the one with the least number of edges.

Dijkstra's Algorithm

- Dijkstra's algorithm is one of the solutions to the shortest path problems. This is a greedy algorithm because from the starting node, traversal is towards the node that is closest. The objective of the algorithm is to find the shortest distance between any two vertices in a graph.
- We begin by giving all edges a distance value, which is the distance of a note from the starting node.
- The distance from the starting point to itself is 0. The distance from the starting node to other nodes are initially unknown and need to be calculated. Therefore we initially assign infinity to these unknown distances. When we calculate the shortest distances using the numbers assigned to the edges, we can update the shortest distances.

Knapsack Problem

- The knapsack problem is a famous problem in computer science. This problem describes an optimization issue.
- Given some values and weights, we have a maximum weight that can go into the knapsack and we want to maximize the weight and at the same time maximize the values.
- That is we want to select items within the weight constrain with the maximum value.
- 0-1 knapsack problem is one where we can not split the item, we must select all of it or not.
- A brute-force solution is when you find all possible subsets within the weight constrain, then pick the subset that has the highest value. The time complexity fo this algorithm would be O(2^n).
- Other solutions with better time complexities are possible. Dynamic programming could be used as well.

The Travelling Salesman Problem (TSM)

- The solution to this problem involves finding the fastest route from one point to the other. Imagine the nodes in a graph as cities and the edges as the roads. We are looking for the optimal route, hence it is an optimization problem.
- This problems falls under a category of problems called NP-Hard, which cannot be solved with a node algorithm in a polynomial time complexity.
- Exact and approximation algorithms can be used to find the exact solution or an approximate solution. Aproximation algorithm use polynomial time. The exact algorithm is a brute-force solution and takes a significantly longer time. Dynamic programming solutions can also give exact solutions.
- One approximate algorithm is the christofides algorithm works by turning a graph into a tree and produces a path that is at least 50% longer than the shortest route.

Data Structures

Collections

- A collection as it's name implies is a group of items. The items don't have to be homogeneous and the items inside a collection are not ordered. List, tuples, and sets are examples of collections

Lists

- A list has all the properties or attributes of a collection. In Python, a list is a collection of ordered items.
- A Python list is mutable, meaning it's size can grow or shrink. That implies items can be added to the list or removed from the list. Items could be replaced as well.
- A list can be homogeneous or heterogeneous

Arrays

- An arrays could be a common implementation of a list. That is, an array is like a list with a few added rules.
- Arrays in Python such as NumPy's arrays or arrays from the array module are homogeneous but a list could be both homogeneous and heterogeneous.
- An array (as well as a list) is a sequence of items stored in a contigous block of memory, meaning that the items are store right after the previous one in the memory block, making arrays to be much faster for operations such as indexing compared to a linked list. The time complexity to index an element is constant or O(1).
- Since the elements of an array (and a list as well) are store in a contigous block of memory, insertion or delete operation can cause a lot of elements to shift which makes this operation to be much slower compared to a linked list.
- Just like a list, Python array is ordered and can be indexed, sliced, concatenated. Arrays also support other sequence operations such as concatenation and multiplication.
- An array is different from a list in terms of operations. For example, element-wise arithmetic operations can be done with arrays but not with a list.

Linked Lists

- A linked list is sequence of objects called nodes containing data and references for the next notes.
- Compared to arrays and lists, nodes in a linked list cannot be accessed by indexing. One must begin from the head or fist node and traverse the list to access a node, which is time consuming compared to indexing a list.
- There are two types of linked list, singly and doubly linked list. A singly linked list allows a node to reference only the next node while a doubly linked list allows a node to reference both the next and the previous node.
- The next attribute of the last node in a linked list points to null or None.
- Items can easily be added and removed from a linked list compared to an array or a list.
- http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson1_1.htm (http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson1_1.htm)

In [12]:
```python
# create a singly linked list in Python using OOP
class LinkedNode():
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

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

# instantiate the nodes
node1 = LinkedNode(20)
node2 = LinkedNode(25)
node3 = LinkedNode(18)

# link the nodes
node1.next = node2
node2.next = node3

def print_list(node):
    while node:
        print(node)
        node = node.next

print_list(node1)
```

```
20
25
18
```

In [13]:
```python
# create a doubly linked list in Python using OOP
class Node():
    def __init__(self, value, before=None, next=None):
        self.value = value
        self.before = before
        self.next = next


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

# instantiate the nodes
node1 = Node(20)
node2 = Node(25)
node3 = Node(18)

# link the nodes
node1.next = node2
node2.next = node3

node3.before = node2
node2.before = node1

def print_next(node):
    while node:
        print(node)
        node = node.next

def print_backward(node):
    while node:
        print(node)
        node = node.before

print_list(node1)
print("\nprint backward")
print_backward(node3)
```

```
20
25
18

print backward
```

```
18
25
20
```

In [14]:
```
# modify the doubly linked list
# link the first node to the third node
node1.next = node2.next
print_list(node1)
```

```
20
18
```

Stack

- A stack is a collection and it is a last in first out (LIFO) data structure where the last element added is the first element that can be retrieved. If you only use the .append(value) and .pop() method of a list, the list acts like a stack.
- So, a stack is like a pile of plates where you can add plates on top of the pile and remove plates from the top of the pile.
- Stacks have two primary methods, the push and pop, to add and remove items respectively.
- Stack is mostly useful when you are interested in the most recent elements such as in a news feed.

In [15]:
```python
# an implementation of stack using a list
class Stack():
    def __init__(self):
        self.items = []

    # method for adding an item to the stack
    def push(self, item):
        return self.items.append(item)

    # a method for removing an item from the list
    def pop(self):
        return self.items.pop()

    # a method for checking if the stack is empty
    def is_empty(self):
        return self.items == []

    # a string representation method
    def __repr__(self):
        return str(self.items)

# instantiate an empty stack object
stack1 = Stack()

# add items to the s
stack1.push(1)
stack1.push(3)
stack1.push(5)

print(stack1)

stack1.pop()
print(stack1)

stack1.push(10)
print(stack1)
```

```
[1, 3, 5]
[1, 3]
[1, 3, 10]
```

In [16]:
```python
# another implementation of stack using a linked list
# first create a node
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None


class Stack():
    def __init__(self):
        self.top = None
        self.size = 0

    # a method that adds a node
    def push(self, value):
        node = Node(value)

        if self.top: # if node exist
            #the created node points to the previous node (.top)
            # oposite to what normally happens in stack
            node.next = self.top
            # .top now points to the new node
            self.top = node
        else:
            # top points to the new node
            self.top = node
        self.size += 1

    # define a pop method
    def pop(self):
        if self.top: # if the last node exists
            value = self.top.value # retrieve the value of the last node
            self.size -= 1 # reset length of stack
            if self.top.next: # if the previous node exists
                self.top = self.top.next # set the previous node as last node
            else: # if the previous node does not exist
                self.top = None # point the top node to nothing
            return value
        return None

    # a method that retrieves the top value
    # without removing it.
    def peek(self):
        if self.top:
            return self.top.value
        return None


stack = Stack()
stack.push(3)
stack.push(4)
stack.push(5)

print("The top value: ", stack.peek())
stack.push(6)
print("The top value: ", stack.peek())
print("The removed value: ", stack.pop())
print("the top value: ",  stack.peek())
print("The size of the stack: ", stack.size)
```

```
The top value:  5
The top value:  6
The removed value:  6
the top value:  5
The size of the stack:  3
```

Queues

- A stack is a collection and it is a first in first out (FIFO) data structure where the first element in the queue is the first element that can be retrieved or removed. Practically a queue is a line of customers waiting for some kind of service.
- There are several queue policies determining who is served first. Some policies are first come, first serve (First in, First Out or FIFO). Other policies are based on prior, which customer or element is important based on some criteria.
- For Python queues, the first element in the queue that has been there for the longest time is removed or retrieved first.
- The oldest element in the queue is called the head while the most recent elment is called the tail.
- When you add an element to the tail, the operation is called **Enqueue** .
- When you remove the head element, the operation is called **Dequeue**
- Peek is when you look at the head element but don't actually remove it.
- The queue data structure can be implemented with a linked list.
- There are two special types of queues called a deque or double ended queue and a priority queue.

- Deques: With a deque, you can enqueue or dequeue from both ends (tail or head). That means, you can treat it like a stack, add elements at one end at remove it only from that end, or it could act like a queue where you add elements at one end (tail) and remove them at the other end (head)
- Priority Queue: With a priority queue, you assign each element a numerical priority when you add the element into the queue. When you deque, remove the elements with the highest priority. This does not follow the FIFO policy. However, if all the elements have the same priority, then the FIFO policy is followed and the oldest element is removed first.

Sets

A set is an unordered collect of unique elements.

Maps

- A map is a key-value structure. It is like a look up table where you use the key to lookup the value. For this reasons, they are also called dictionaries. With real-life dictionaries, you use a word (key) to look up the definition (value) of the word.
- Generally, the keys of a map are sets because they are unique. "Keys views are set-like since their entries are unique and hashable": https://docs.python.org/release/3.3.0/library/stdtypes.html#dict-views (https://docs.python.org/release/3.3.0/library/stdtypes.html#dict-views)

Hashing

- Data structures that employ a hash function allows you to do look up in constant time. In a list or set, for example, you need to look through all the element (linear search) to find the one you are looking for.
- The ability to do constant time lookup or search makes your algorithm much faster.
- The purpose of a hash function is to transform some values into ones that can be stored and retrieved easily. The function takes some input value and returns a hash value.
- Hashing is the concept of converting data of an abitrary size to a fixed size.

In [17]:
```python
# for example, we can convert strings to numerical values
ord("H")
```
Out[17]:
```
72
```

In [18]:
```python
# use the map function to convert characters in a string at once
list(map(ord, "Hello"))
```
Out[18]:
```
[72, 101, 108, 108, 111]
```

In [19]:
```python
# we can sum all the values of the characters to get a
# a value for the string "Hello"
sum((map(ord, "Hello")))
```
Out[19]:
```
500
```

In [20]:
```python
# the issue is that if the same words in the string are
# mixed up, you still get the same numerical value
sum((map(ord, "olleH")))
```
Out[20]:
```
500
```

A hash function

- To solve the problem of having the same value when characters are ordered differently, we can multiple the characters by numerical values corresponding to their positions. Storing the text in hash functions can help us to look up the text in constant time. The hash value is used as the index of an list and these indexes can be used to lookup the values in a hash table.

In [21]:
```
def hashing(text):
    values = list(map(ord, text))
    mult = range(1, len(values)+1)
    mult_values = [mult*value for mult, value in zip(mult, values)]
    return sum(mult_values)

print("Hello: ", hashing("Hello"))
print("olleH: ", hashing("olleH"))
```

```
Hello:  1585
olleH:  1415
```

In [22]:
```
# we could still run into issue of havig the same value
# for different text but this can be fixed
hashing("ad"), hashing("ga")
```

Out[22]:
```
(297, 297)
```

Collision

- A collision is when a hash function generates the same value given different inputs.
- The way to fix this is to change the value of the hash function or to change the hash function completely. The value of the hash function can be replace the original input values but this can take up more space as well especially if the input values were large. You may have to choose a hash function that spreads out your values nicely but uses a lot of space. You could also use a hash table that uses less buckets but would need to do some searching inside the bucket.

Hash Maps

Maps have keys and values and you can use the keys as input into a hash function, then store the key-value pair to in the bucket of the hash value produced by the function.

Trees

- Trees are hierarchical data structures. Trees have roots and leaves. A tree has a parent-child relationship between items. A collection of trees is called a forest.
- A tree is an extension of a linked list. The root is rather the top node and all the nodes must be connect with no cycles.
- A tree could be described in terms of levels or how many connection it takes to reach the root. The node at the root is at level 1. Nodes at lower level are parents of nodes at the higher level.
- Children are only allow to have one parent but a parent can have two or more children and these children are called siblings.
- You can also call a node at a lower level ancestor and node at higher level descendants.
- Nodes at the end that don't have any children are called leaves or external node.
- A parent node is called an internal node
- A connection or edges or arrow linking a parent and a child. Many connections together make a path.
- The number of connections is the depth of the tree.
- The height of a tree is overall is the height of the root node. The height of a node is the number of edges between it and the furthes leave on the tree.
- A leave has a height of zero and a parent of a leave has a height of one.

Tree traverse

- Traversal is the idea of accessing the elements of a data structure. Traversal in a tree is not linear and there is no clear way to traverse a tree. It is complicated.
- There are two major ways to traverse a tree called depth-first search (DFS) and Breath-first-search (BFS).
- DFS: Exploring children node is the priority. There are different approaches to DFS. Pre-Order traversal means you need to check off the nodes before visiting their children. Start at the root, and traverse down until you reach a leave, then go back to parent of the leave, then to the other slibling leave. Travel back to the root and check the right child and continue down the leave. There is also In-Oder traversal for DFS. We only check off the node when we see the left child and move back to it. With Post-Order, we don't check off the child until we have visited all it's dependents.
- BFS: The priority is visiting every note on the same level before visiting their child nodes. It is level order.

Binary Tree

- Binary trees are simply trees where the parents have at least two children. That means notes can have zero, one, or two children. We can use any of the traversal algorithms to traverse the tree.
- To delete: you can delete a leave directly. If you are deleting an internal node, you may end up with a gap. If a parent has one child, you can link the child to its grand parent.

- If a node has two children, you could promote one of the children up (to replace the parent).

Binary Search Tree (BST)

- this is a type of special type of binary tree but the rule is that every value on the left is smaller than the values on the right and every value towards the root are smaller and values towards the leave are bigger. So, the values are sorted. This makes it easy to search and the runtime for a search is O(log(n)). Deleting is same as for a generic binary tree.

Heaps

A heap is another specific type of tree with some of it's own additional specific rules. The elements are arranged in a decreasing or increasing order. So there is max heaps (parents have higher value) or min heaps (parents have lower values). A binary heap must be a complete tree which means except the leave, all nodes must be full (have two children).

Graphs

- A graph is a data structure designed to show relationships between objects. It is also sometimes called a network. The purpose of a graph is to show how things are connected together.
- A graph is similar to a tree in many ways. A tree is actually a specific type of graph. Cycles are possible in graphs so graph don't have root nodes. The nodes called vertices, are objects that store data and the edges are connections between objects. Edges could contain data too, for example nodes could be people and edges could be how many times they worked on a project together.
- Directed graphs are graphs where edges or connections have direction. For example, you may have cities connected and directions are indicated in the edges to show whether your trip from one city to the other was a round trip or not.
- Undirected graphs have edges with no sense of direction. For example, a graph that is a network of friends may have no direction.
- Note that cycles in graphs when writing algorithms can be problematic because you may end up with an infinite loop. Directed Acyclic Graphs (DAG), that is directed graphs with no cycles is commonly used.
- The study of graphs is called graph theory.
- Connectivity is a property of a graph. A graph could be connected or disconnected. connected graphs have all vertices connected and disconneted graphs have at least a vertice that is not connected to other vertices.
- Connectivity helps us to understand how many elements can be removed for a graph to become disconnected and we can use connectivity to tell which graph is stronger.

Graph Representations

- Graphs could be functionally the same, but could be built in different ways.
- OOP could be used where vertex and edge objects are created and connected. However, it could be slow or inconvinient to traverse through the objects.
- An edge list could also be used where a nested list containing a list of two numbers stores two numbers, representing the nodes that are connected by edges.
- An adjancency list could be used. This is also a nested list of list where the index of each list is a node and the elements inside each list are the nodes connected to the one represented by the index.
- Graph traversal is almost the same as a tree traversal and they are easy to traverse based on connections. There are two basic methods for traversal, the Depth-First Search (DFS) and Breath-First Search (BFS) just as for tree traversal.
- DFS: Note that trees have roots but graph don't. When you traverse the node, mark it as seen. Start by picking a node, mark it as seen, then go through an edge, mark it as seen and this is implemented by storing the node to a stack. Keep moving towards nodes and edges that have not been seen. When you run out of new notes, pop the current node and go to the one right before it. Continue this process until find the node you are looking for. The run time is O(|E| +|V|) where E is number of edges and V is number of vertices.
- BFS: Start with first node, mark it as seen by adding it to a Queue. Then visit the adjacent node and mark them as seen and add them to the queue. If you run out of nodes, deque or remove the head (oldest elment) and start at that point again and move to other directions not yet traverse. Repeat the process until you find the element you are looking for. This process is almost like creating a tree from a graph, the first node in the queue becomes the tree root. The runtime is O(|E| +|V|).

Eulerian Paths

- There different notable paths that could be traverse in a graph. One is the Eulerian path, named after the mathematician Euler. You start at one node, traverse through all edges and might end up on a different node. However, in an Eulerian cycle, you traverse each edge only once and ends up on the same node where you started.
- Graphs can only have Eulerian cycles if all the nodes are at the same degree, that is have the same number of edges. The runtime for traversing Eulerian path is O(|E|). There are also other paths such as the Hamiltonian path where every edge is traversed.