Data Structures & Algorithms

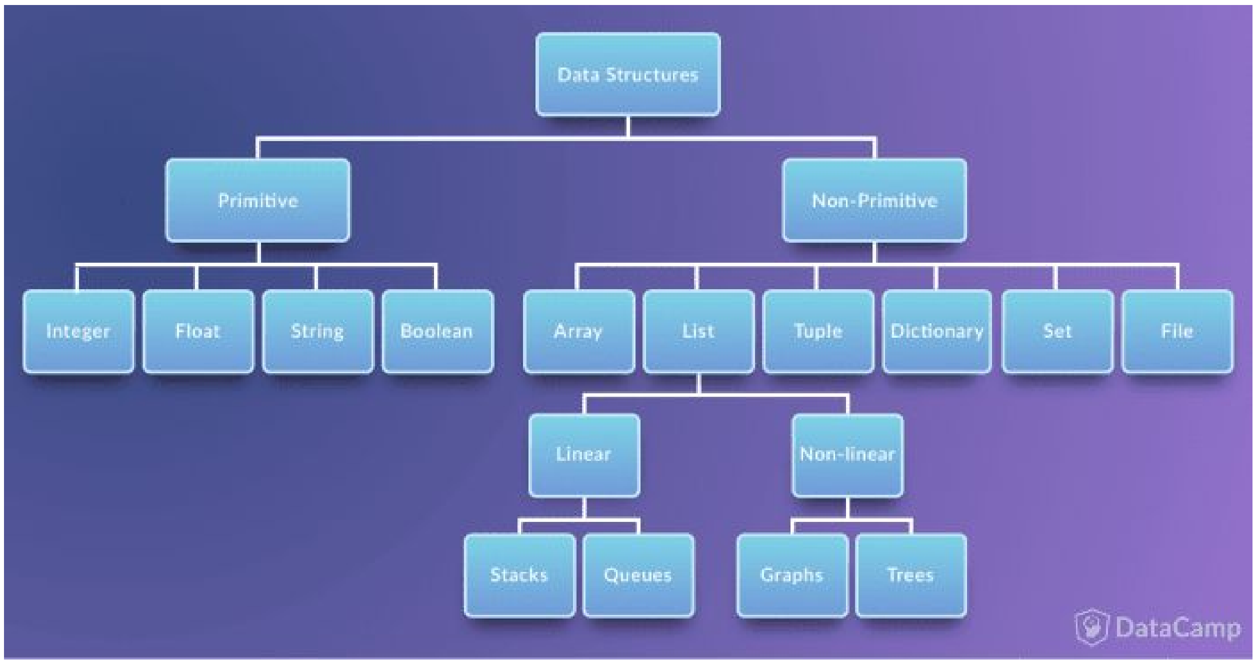
Data Structures

Themes

- ❏ Lists
- ❏ Maps
- ❏ Sets
- ❏ Stacks
- ❏ Priority Queues
- ❏ Bags
- ❏ Binary Trees
- ❏ Tries

- ❑ Graphs
- ❑ Heaps
- ❑ Hash Tables

Concrete Examples: LinkedList, ArrayList, Vectors



Python

Strings

Strings are immutable. Functions:

- "44".isdigit()- is the string made of digits only ?
- "44".isalpha()- is the string made of alphabetic characters only ?
- "44".isalnum()- is the string made of alphabetic characters or digits only ?
- "Aa".isupper()- is it made of upper cases only ?
- "aa".islower()- or lower cases only ?
- "Aa".istitle()- does the string start with a capital letter ?
- text.isspace()- is the string made of spaces only ?
- count(elem) - count of elem
- mystr.title()- return a titlecase version of the string
- mystr.capitalize()- return a string with first letter capitalised only.
- mystr.upper()- return a capitalised version of the string

- mystr.lower()- return a copy of the string converted to lower case
- mystr.swapcase()- replace lower case by upper case and vice versa
- mystr.center(40)- center the string in a string of length 40
- mystr.ljust(30)- justify the string to the left (width of 20)
- mystr.rjust(30, '-')
- mystr.strip(); mystr.rstrip(); mystr.lstrip()
- mystr.replace('dummy', 'great', 1)- the 1 means replace only once
- mystr.translate(None, 'aeiou')
- mystr.endswith('ing')- may provide optional start and end indices
- mystr.startswith('This')- may provide start and end indices
- mystr.find('is'[,index])- returns start index of 'is' first occurence
- mystr.rfind('is')- returns start index of 'is' last occurence
- mystr.index('is')- like find but raises error when substring is not found
- mystr.rindex('is')- like rfind but raises error when substring is not found
- message = ' '.join(['this' ,'is', 'a', 'useful', 'method'])
- message.split(' ')
- 'this is an example\n of\ndummy sentences'.splitlines() - separates by ‘\n’

Lists

GENERAL

In Python Lists and tuples are like arrays.Tuples like strings are immutables. Lists are mutables so they can be extended or reduced at will.

USE CASES

Tuples are faster because they are immutable and consume less memory.

Tuples can be used as keys on dictionaries; key-value pairs in dictionaries; tuple unpacking (x,y,z) = (1,2,3); swapping (a,b)=(b,a).

Operation	Running Time		Operation	Running Time
len(data)	O(1)		data[j]=val	O(1)

data[j]	O(1)		data.append(val)	O(1)*
data.count(val)	O(n)		data.insert(k,val)	O(k-n+1)*
data.index(val)	O(k+1)		data.pop()	O(1)*
value in data	O(k+1)		data.pop(k) del data[k]	O(k-n)*
data == data2 !=;>;<	O(k+1)		data.remove(val)	O(n)*
data[j:k]	O(k-j+1)		data1 += data2	O(n_2)*
data+data2	O(n_1+n_2)		data.reverse()	O(n)
c*data	O(cn)		data.sort()	O(nlogn)

List and Tuple   List

HOW TO

Lists are enclosed in brackets:

l = [1, 2, "a"]

Tuples are enclosed in parentheses:

t = (1, 2, "a")

List functions:

- append - to add one elem to end
- extend to add list to the end
- index(elem, [start index], [stop index]) - search for the index of elem, error if not found
- insert(index, elem) - inserts elem at exact location
- remove(elem) - remove the elem from list
- pop() - removes the last elem
- count(elem) - count the number of elements
- sort() - sorts the list; also sort(reverse=True)
- reverse() - reverses the list
- l[::2], l[0:9:2] - second dot indicate steps
- l2 = l[:] -> is a copy; l2 = l -> is a reference; import copy; l2 = copy.deepcopy(l) -> copies nested lists, otherwise they are also references

Tuple functions:

- += - to add elem (as in lists)
- *= - repeat tuple by multiplying (as in lists)
- index(elem) - to find an index of the elem
- count(elem) - to find the count of the elem
- t2 = t -> is a copy (different from lists) however does not copy nested lists
- str(t) - stringifies tuple

Linked Lists

GENERAL

Has data, and next to point to the next element. Doubly Linked Lists also have a previous link to traverse both forward and backwards.

USE CASES

HOW TO

```
class SinglyLinkedListNode:

    def __init__(self, node_data):

        self.data = node_data

        self.next = None


class SinglyLinkedList:

    def __init__(self):

        self.head = None

        self.tail = None
```

Maps

GENERAL

A dictionary is a sequence of items. Each item is a pair made of a key and a value. Dictionaries are not sorted. No duplicate keys.

you can compare dictionaries! Python first compares the sorted key-value pairs. It first sorts dictionaries by key and compare their initial items. If the items have different values, Python figures out the comparison between them. Otherwise, the second items are compared and so on.

HOW TO

Dictionaries are built with curly brackets:

```
d = {"a":1, "b":2}
```

Functions:

- keys() - returns keys as a list
- values() - return values as a list
- items() - return list of key/value tuples
- has_key(key) - checks if key exists return true if so/ key in d
- get(key) - return the value / d[key]
- pop(key) - return but also removes the value
- popitem() - removes and returns a pair (key, value); you do not choose which one because a dictionary is not sorted
- copy() - to copy dictionary
- clear() - remove items
- del d[key] - remove one item
- d1.update(d2) - add all pairs of key/value from d2 into d1
- sorted()/reversed() - to sort/reverse

Sets

GENERAL

Sets are constructed from a sequence. Sets cannot have duplicate values. Just as with dictionaries, the ordering of set elements is quite arbitrary, and shouldn't be relied on.

USE CASES

Since sets cannot have duplicate values, they are usually used to build a sequence of unique items (e.g., set of identifiers).

HOW TO

Sets are built with set function:

```
s = set([1,2,3,4])
```

Functions:

- a | b - Union of two sets
- a & b - Intersection
- a < b - checks if a is a Subset of b true if so
- a - b - finds Difference between sets
- a ^ b - Symmetric Difference includes both differences
- copy() - to copy a set

Stacks

USE CASES

A stack can be used as a general tool to reverse a data sequence; to match Parentheses and HTML Tags

Time complexity - O(1)

The implementations for *top*, *is_empty*, and *len* use constant time in the worst case. The O(1) time for *push* and *pop* are amortized bounds, but there is occasionally an O(n)-time worst case, where n is the current number of elements in the stack, when an operation causes the list to resize its internal array.

Functions: push, pop

If implemented with linked lists, it has a head.

HOW TO

Lists can be used as stacks LIFO.

Functions - append(elem), pop()

Priority Queues

USE CASES

used by many computing devices, such as a networked printer, or a Web server responding to requests.

When pop is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the pop. Therefore, a call to pop(0) always causes the worst-case behavior of O(n) time.

Functions: enqueue, dequeue, peek

If implemented with linked lists, it has a head and a tail. Nodes have value, next and priority. When dequeue, remove the element with the highest priority. If elements have the same priority, the oldest got to be dequeued.

HOW TO

Lists can be used as queues FIFO. Functions - append(elem), pop(0); a call to pop(0) always causes the worst-case behavior of O(n) time.

Internally, the queue class maintains the following three instance variables:

data: is a reference to a list instance with a fixed capacity.

size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).

front: is an integer that represents the index within data of the first element of the queue (assuming the queue is not empty).

We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the front index to zero.

Deque is a double ended queue, which can be used for both stack and queue. Python has a library for deque to be used for queues.

Functions - append(element), popleft()

```
>>> from collections import deque

>>> queue = deque(["Eric", "John", "Michael"])

>>> queue.append("Terry")           # Terry arrives

>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
```

Trees

GENERAL

The root is often called the parent and the nodes that it refers to below are called its children. The nodes with the same parent are called siblings.

Height - # of edges between it and the furthest leaf. Height of a leaf - 0, its parent is 1. Height of tree is the height of a root.

Depth - # edges to the root. Height and Depth should move inversely.

Binary tree - a parent has at most 2 children.

Binary Search Tree - sorted; left nodes are smaller, rights are larger.

BSTs can be balanced and unbalanced. Distribution is skewed to either right or left side. This structure can start from the root or the subtrees. Worst Case scenario for BSTs. Every action is linear.

Heap - increasingly/decreasingly sorted trees where root is either the max or the min of all elements. Heaps do not need to be binary trees.

Search O(n), but we can stop searching im max heap if our element is bigger than the nodes.

Max heap - parents should always have value greater than its children. Peek function(to see the max elem) is O(1) since root is first.

Min heap - parents should always have value less than its children

Heapify - operation in which we reorder the tree based on the heap property(max/min). Insert the element to the empty spot, keep comparing it with its parent and swap when necessary. The same with extracting, put the rightmost to the deleted spot and start comparing its children and swap.

Insert and delete - O(logn) (Worst Case, moving it all the way up/down the tree)

Self-Balancing Tree - minimizes the levels it has so that it will be balanced tree. The most common example is red-black tree(extension of bst).

Rule 1. Nodes are assigned an additional color property-red/black.

Rule 2. Existence of null leaf nodes.Every node is does not have two leaves must have null children(black color).

Rule 3. If a node red, both of its children must be black.

Rule 4.(optional) Root node must be black.

Rule 5. Every path from a node to its descendant null nodes must contain the same number of black nodes.

Insert only red nodes, then change the color if needed.

If the parent and its sibling are both red, change them to black and make grandparent red.

If parent is red and its sibling is black, rotation is needed

Let’s say a node is a right child but the parent is a left child. Perform a left rotation - shift one place to the left while maintaining positions.

Let’s say a node and a parent is a right child. Perform a right rotation - shift one place to the right while maintaining positions.

Change color in rotation if needed.

Insert, search, delete - O(logn) average and worst cases. Delete is not O(n) because it is always balanced.

USE CASES

Tree - If the tree does not have an order, search and delete is O(n);

BT - Inserting is done when we find an empty spot. In the worst case, it can result in traversing to the end-height. Height changes by level in the binary tree, adding $2^{level-1}$ number of nodes. Insert O(logn)

BST - since order exists, search and insert- O(logn) averagely, in the worst case O(n); delete still as in ordinary trees

HOW TO

Heap can be implemented with arrays.

Graphs

GENERAL

A graph is designed to show relationships between objects; They are networks consisting of nodes, also called vertices which may or may not be connected to each other. The lines or the path that connects two nodes is called an edge(can have weights). If the edge has a particular direction of flow, then it is a directed graph, with the direction edge being called an arc. Else if no directions are specified, the graph is called an undirected graph.

Common graph - DAG (Directed Acyclic Graph)

Connectivity - describes the minimum number of elements removed for a graph to be disconnected. (If each vertex has just one edge, removing 1 edge would be enough to disconnect the graph)

Traversal

DFS - follow 1 path as far as it goes.

BFS - look all the nodes adjacent to one, before moving on to the next level.

Eulerian Path - a path which traverses through an edge at least once.

Eulerian Cycle - traverse every edge only once and end up at the same node you started with. Possible if all vertices have an even degree/even number of edges connected to them.

Hamiltonian Path - go through every vertex once.

Hamiltonian Cycle - start and end at the same vertex.

USE CASES

trying to find if there exists a path between two nodes, or finding the shortest path between two nodes, determining cycles in the graph.

BFS - Solution to the shortest path problem on an unweighted graph.

HOW TO

Dictionaries can be used to build a graph.

Edge Lists can be used to represent a graph - [[0,1],[1,2],[1,3],[2,3]] - 4 edges with id number for vertices

Adjacency List is another way to represent a graph - [[1],[0,2,3],[1,3][1,2]] - index of a vertex corresponds to a list of vertices it is adjacent to

Adjacency Matrix can be also used. 2D array but the length inside is the same. Indices in the outer array represent id of vertices. List inside represents vertices. 1 means there is an edge between vertices, 0 means no. [[0,1,0,0],#zero [1,0,1,1], #one [0,1,0,1], #two [0,1,1,0] #three]

DFS implementation - use stacks to store the node you have seen. If you hit a node you have seen before, go back and try another edge. If you are out of edges with new nodes, you pop the current node, go back to the one before it. Continue, until you popped everything out of the stack or found the node you were looking for.

DFS implementation - use recursion. pick an edge, mark the node as seen until you run out of new nodes to explore. *Time Complexity* - O(|E|+|V|)

BFS implementation - use queues. add adjacent nodes to the queue as you see them. When run out of the edges, dequeue a node and use that as a next starting place. *Time Complexity* - O(|E|+|V|)

Algorithm to find Eulerian Cycles: *Time Complexity* - O(|E|)

1.Start at any vertex, follow edges until you return back to that vertex

2.If every edge was not encountered, start from an unseen edge of your visited vertex

3.Perform the first step and create another path. Continue 2 and 3 until every edge is seen.

4.Add the paths together combining at the nodes they have together.

Hash Tables

GENERAL

Lookup times O(1)

Hash function - common way is to take last two digits and divide by a fixed number, the remainder will be the index in the array.

Collision Resolve - change hash function (bigger space, moving to the new array is slow) or change array structure to have buckets(list)(iterate within each bucket O(m))

Load factor - # of values/ # of buckets

String Keys - using ascii values(ord() function to change letter to number)

30 or less words - use first letter

Formula - s[0]*31^(n-1)+s[1]*31^(n-2)+...+s[n-1]

words with 3 or 4 letters - will have huge hash values.

Algorithms

Themes

- ❏ Greedy Algorithms
- ❏ Divide and Conquer
- ❏ Dynamic Programming
- ❏ Recursion

❑ Brute Force Search

Concrete Examples: Breadth First Search, Depth First Search, Binary Search, Merge Sort, Quick Sort, Tree Insert/Find, A*

Extra Concepts: Bit Manipulations, Singleton Design Pattern, Factory Design Pattern, Memory(Stack vs Heap), Recursion, Big-O Time

Big-O Time

Our seven functions are ordered by increasing growth rate in the following sequence:

1, logn, n, nlog n, n2, n3, 2n.

Breadth First Search

GENERAL

Priority is to explore every node on the same level.

Depth First Search

GENERAL

Priority is to explore children.

Approaches of DFS

```
        B

  A           C
```

1. In-order: left first, then root, then right A B C; allows nodes to be in order
2. Pre-order: root first, then left and right, B A C;
3. Post-order: left first, then right then root A C B;

Binary Search

USE CASES

efficiently locate a target value within a sorted sequence of n elements. find a median, if target is less work with first part, else if larger with second part, if equal stop. Divide and Conquer.

Time complexity - O(logn);

HOW TO

```
def binary search(data, target, low, high):
”””Return True if target is found in indicated portion of a Python list.

The search only considers the portion from data[low] to data[high]
inclusive.
”””
if low > high:
      return False # interval is empty; no match
else:
      mid = (low + high) // 2
if target == data[mid]: # found a match
      return True
elif target < data[mid]:
      # recur on the portion left of the middle
      return binary search(data, target, low, mid - 1)
else:
      # recur on the portion right of the middle
      return binary search(data, target, mid + 1, high)
```

```
def binary_search(input_array, value):

    low = 0

    high = len(input_array)-1


    while(low <= high):

        mid = (high+low)//2

        if input_array[mid] == value:

            return mid

        elif input_array[mid] < value:

            low = mid+1

        elif input_array[mid] > value:

            high = mid-1


    return -1
```

Recursion

USE CASES

Fibonacci

You may have noticed that this solution will compute the values of some inputs more than once. For example get_fib(4) calls get_fib(3) and get_fib(2), get_fib(3) calls get_fib(2) and get_fib(1) etc. The number of recursive calls grows exponentially with n.

In practice if we were to use recursion to solve this problem we should use a **hash table** to store and fetch previously calculated results. This will increase the space needed but will drastically improve the runtime efficiency.

HOW TO

```
def get_fib(position):

    if position == 0 or position == 1:

        return position

    return get_fib(position - 1) + get_fib(position - 2)
```

Bubble Sort

GENERAL

Bubble Sort is a naive approach. Compares adjacent values, switches if the first one is larger than the second. In-place sorting algorithm, since no need for an extra space. n comparisons for n steps

USE CASES

Time complexity - O(n^2)(Worst and Average Case)

Time complexity - O(n) If array is already sorted (Best Case)

Space complexity - O(1)

Merge Sort

GENERAL

Divide and conquer.Break up array,sort and build up again. As we do this, we use new arrays so space complexity will be bigger than bubble sort. n comparisons for logn steps, which is less than bubble sort.

USE CASES

Time complexity - O(nlogn)

Space complexity - O(n)

Quick Sort

GENERAL

Continuously choosing pivots swap all values less than it to the left and more than it to the right till the array is sorted. In-place sorting algorithm.

If the array is nearly sorted, do not choose this sorting!!!

USE CASES

Time complexity - O(n^2)(Worst Case)

Time complexity - O(nlogn)(Average and Best Case)

Space complexity - O(1)

Dijkstra’s Algorithm

USE CASES

Finding the shortest path on weighted undirected graphs.

Time complexity - O($|V|^2$)(Worst Case)

Time complexity - O(|E|+|V|log(|V|))(If the priority queue is implemented efficiently)

HOW TO

Begin with giving vertices distance values - sum of edge values to the node; start with the infinity as a placeholder.

Common implementation - min priority queue. Continue extracting the minimum of the queue, exploring adjacent nodes, until the node we are looking for is extracted from the queue/everything else has a distance of infinity which means a path does not exist. Because of choosing a min distance value to continue with, Dijkstra is called the Greedy Algorithm. Update distance value in the queue if the value can be decreased.

Dynamic Programming

GENERAL

Solving the problem first for the trivial case and storing the solution in a lookup table. Also, using the equation at each step when complexity is added. The equation usually has the value *pre-computed and stored in the table-memoization*.

Knapsack Problem

USE CASES

Time complexity - O(2^n)(Brute Force method)

Time complexity - O(nW)(Faster way in How to; n-# of element, W-max limit of knapsack)

HOW TO

Create an array to store max possible value for every weight up until our max weight. Take the first element(or sort and take smallest,i guess?) and update the index in the array of its weight by the value. For example, object weight:2,value:6 -> update array[2] = 6. Also, update everything after, as well till max limit.

Looking at the next object, if the value on the index is smaller than the object, update it till the max limit. For example, object weight:5,value:9; 6<9 -> update array[5] = 9…

If the value in the array is bigger than the value, leave it. For the next value in the array, compare the value in the array and the value of the object + the first value in the array. For example, object weight:4,value:5; 1+4 = 5 -> array[5] > array[4]+array[1] do not update array[5]; 2+4 = 6 -> array[6] < array[4]+array[2] update array[6]