FIT 1008

Algorithmics...

Algorithms:

- Well specified
- Defines input-to-output relationship
- Various problem instances exist

To solve an algorithm:

- A finite set of instructions
- Correct: Halts for any problem instance, correct input-to-output relationship

Eg: Given (+ve int. k, determine whether k is even.

- is_even(10) → True
  is_even(7) → False

- Assume <u>binary representation</u> of input k.

- The right most bit: If 0 → even, If 1 → odd

Algo 1: Traverse the bits from left to right.

- Assume that we have access only to the left-most bit of k. (Unrealistic assumption though).

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

ie. Let k = 5
k = 0b01
b = 1
res = False

OR

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res.
```

⟶ FASTER ALGORITHM

Eg 2: Alice thinks of some num. $0 \le a < k$. The value of a is hidden from us. Given the known upper bound k, the problem is to find a.

Protocol to follow.

- Iteratively make guesses $b \in \{0, \ldots, k\}$
- Alice replies with -1, 0, or 1 if $a < b$, $a = b$, $a > b$ respectively.
- Problem solved when guess is correct ($a = b$)

Implementation:

```
def alice_replies(b):
    if b < hidden_value_A:
        return -1
    elif b > hidden_value_A:
        return 1
    else:
```

Algorithmic / Computational Complexity:

Time Complexity: How much time does an algorithm spend solving the corresponding problem. (measured as the no. of "elementary operations" performed

Space Complexity: How much space does an algorithm spend solving the corresponding problem. (measured as the amount of "memory" occupied simultaneously

↳ Out of Scope for FIT1008!

BOTH types of complexity are measured with regards to the input size

Time complexity as a function of size:

- An algorithm's time complexity can be measured as a function of the input size
- Given an input size of n, we can calculate the value $T(n)$ as the number of "elementary" steps (instructions) performed by the algorithm on the input of size n
- Each elementary step contributes a 1 to the total value $T(n)$

↳ Often they are memory accesses

Main Concern: How does the algorithm perform when n is large [How does it scale]

↳ Here, we consider the Order of approximation of $T(n)$

↳ This is where the Big-O notation helps

When $n \to \infty$, we apply asymptotic analysis for this.

What is Input Size?

Numeric input: Length of its binary representation

Collection (Array, list, set, stack): Number of elements in collection

String input: Number of characters

nxm matrix: Number of rows (n) & number of columns (m)

Graph or Tree: Number of nodes

Asymptotes and Big-O Notation:

Rationale:

- In practice, we don't count the exact number of elementary steps as it is difficult, time consuming & error prone
- Instead, we determine the most "expensive" parts of the code
  - ↳ We analyse the impact on the "rate of growth" of the algorithmic complexity
    - ↳ This is where Big O notation comes in handy

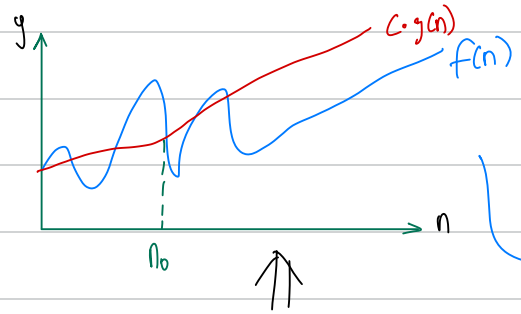
What is O()?:

- Assume we are given 2 functions f(n) & g(n), f(n) and g(n) $\in \mathbb{N}$
- If $f(n) = O(g(n))$ if the following holds:
  - There are 2 (pos) constants $n_0 \& c$ such that all $n \geq n_0$.

    Ans: $0 \leq f(n) \leq c \cdot g(n)$
  - g(n) is the asymptotic upper bound for f(n). i.e as $n \to \infty^+$

    ↳ This is what we call asymptotic analysis.

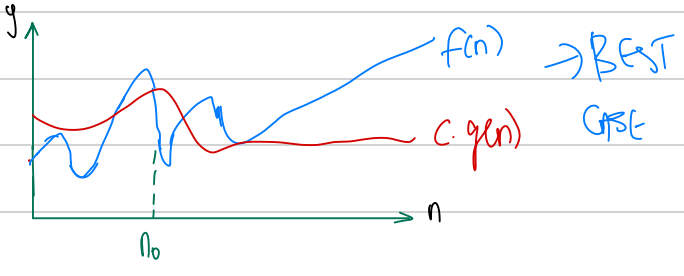
Eg:



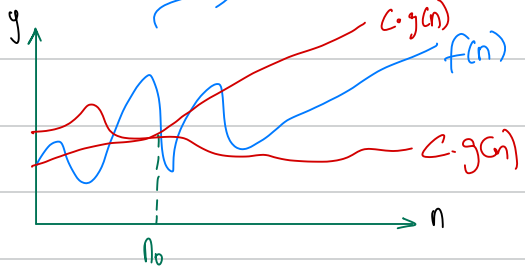
$f(n) = O(g(n))$

↳ WORST CASE

Big O provides the asymptotic upper bound for f(n)



→ BEST CASE



→ MIDDLE CASE

$f(n) = \Omega(g(n))$

This provides asymptotic lower bound for f(n)

$f(n) = \Theta(g(n))$

This provides asymptotic tight bound for f(n)

Common functions for growth rates (Big O): e^n, n^2, $n\log n$, $\log n$, $\sqrt{n}$, n, 1...

Eg: Is $4n^2 + 1000n + 100 = O(n^2)$

Ans: YES. Let $c = 20$. Then $4n^2 + 1000n + 100 \le 20n^2$ for all $n \ge n_0 = 13$

Eg: Is $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$

Ans: NO. Let $c = 2$, then $n^3 + n^2 \log^2 n \le 2n^3$ for all $n \ge n_0 = 16$.
↳ w/o such constant c' & n_0'.

In general, we can focus on a single component growing faster than others.
Such a component dominates all the others.

BASIC Big-O PROPERTIES:

- If c is any positive constant, then $c = O(1)$
- Sum of $O()$: If $f_1(n) = O(g_1(n))$ & $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
  pick fastest growing component ↑ the dominating one
- Product of $O()$: If $f_1(n) = O(g_1(n))$ & $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

  In particular, if $f_1(n) = O(g_1(n))$ & $f_2(n) = O(1)$, then
  $f_1(n) \cdot f_2(n) = O(g_1(n))$ → may ignore constant.

Sorting:

Sorting Lists: (increasing order)

Input: A list (not necessarily sorted) of 'orderable' element types
Output: A list with the same elements as the input BUT sorted in increasing order

Bubble Sort:

- Start with the left most element X
- Compare X to the element to the element Y on its right
- If X>Y, swap them, otherwise don't
- Move one position to the right.

Invariant: property that remains unchanged.

In BubbleSort, there are many invariants. Also, in each traversal, at most n-1 swaps are performed, where n is the length of the list.

One invariant is particularly interesting:
After every traversal, the largest yet unsorted element gets to its final place. Tells us max. no. of traversals needed to sort.

Bubble Sort Optimisation:

- Use the invariant: After every traversal, the largest unsorted element gets to its final place
- Each iteration can avoid comparing the last element if moved as it is already in its final place
- We can do this by marking the last element that might need to be swapped
- Everything to the right of blue mark is sorted, this blue mark keeps moving left.

More optimisations:

Use boolean swapped initialised to false
- Set to true every time there is a swap.
- If after one iteration not swapped is true, stop!

Bubble Sort incrementation:

- A sorted algorithm is incremental if there is a sorted list and one new element
- Uses one (or a few) iterations of the algorithm to return a sorted list that has the new element
- "After one iteration, everything is sorted" – Very incremental

- Is Bubble Sort stable: It is, IF it maintains the relative order among elements (relative order must be same from a list & another new list).

	ORIGINAL	NEW 1	NEW 2
ie	1, 3, 5, 9	3, 5, 7, 9	1, 5, 3, 9
	3 before 5!	STABLE, 3 before 5	UNSTABLE, 5 before 3

SELECTION SORT:

- No need to sort so many swaps
- Start at leftmost unsorted element ↓, set it as current minimum ↓
- Traverse, ↓, the rest & find the minimum element ↓ in the rest of the list (if different from current one)
- ~~If~~ Minimum element gets pushed to the final place

Selection Sort Optimisation: Not easily as we do not know the relative order

Selection Sort Incrementation: Not incremental

Selection Sort Stability: Not stable

Insertion Sort :

- Split list into two parts, 1 sorted, 1 unsorted.

ie Part S → already sorted [initially one element]
Part U → Unsorted

Extend S by taking any element from U & inserting it in S maintaining the order

Invariant: elements in S (left of split) are sorted and grow by 1 in each iteration

∴ No. iterations required: n-1

BUT elements in S might not yet be in their final position as other elements from the list is inserted

*. Store unsorted in temp, Shift bigger to right, store temp into freed

Insertion Sort incrementality: Yes as the existing list will be considered as sorted already (Thanks to the invariant)

Insertion Sort Stability: Yes as each temp element shuffles the original list but the actual shuffle maintains its relative order.

Data Types...

Abstract Data Types:

- An abstract data type provides information regarding:
  - The possible values of the type and their meaning
  - The operations that can be done on them

  But NOT on its implementation (ie how):
  - The values are stored
  - The operations are implemented

- Users interact with data only through provided operations

Gives of ADT's:

- Build programs without knowing their implementation (simplicity)
- The implementation can change without affecting you (maintenance)
- If several ADT's available, you could easily use any (flexibility & reusability)
- Different compilers can use different implementations (Portability)

- At some point, ADT's have to be implemented.

- A data structure provides:
  - A particular way in which data is physically organised (so that certain operations can be performed efficiently)

  Eg: The array data structure:
  - fixed size
  - Data items are stored sequentially
  - Each item occupies the same amount of space

  } Physical organisation

- Primitive (built-in) data types OR user-defined data types
- Simple data types OR complex data types (single data vs. multiple data fields)

Abstract Data Types: provides information about:
- possible values for that data type
- Meaning for those values
- Operations that can be done on them

Data Types: provide the same info as ADT's plus:
- The way those values are implemented

Data Structures: provide information about:
- the way in which data is physically organised in memory
- the only operation it provides is access.

Stack Abstract Data Type:

Two main factors:

1. Its operations follow a Last In First Out (LIFO) process
   - Last element added is first element deleted.

2. Access to any other element is Unnecessary (and thus not allowed)
   - If you need to access another element, you need to choose a different ADT

Key operations:

Create a stack (Stack)
Add an item to the top (push)
Take an item of the top (pop)

Other common operations include:
- Look at item on top w/o altering stack (top/peek)
- What is its length?
- Is the stack empty?
- Is the stack full?
- Empty the stack (clear)

Data Structure with Arrays:

Base Stack Class:

```
From abc import ABC, abstractmethod
From typing import TypeVar, Generic

T = TypeVar('T')

Class Stack (Generic[T]):

    def __init__(self) -> None

        Self.length = 0

    @abstractmethod
    def push(self, item: T) -> None:

        pass

    @abstractmethod
    def pop(self) -> 'T':

        pass

    @abstractmethod
    def peek(self) -> T:
        pass

    def __len__(self): -> int:

        Return self.length

    def is_empty(self) -> bool:
        return len(self) == 0

    @abstractmethod
    def is_full(self) -> bool:
        pass

    def clear(self) -> None:

        Self.length = 0
```

Big O for this function:

__init__, __len__ & clear are O(1) (Always constant)

is_empty is O(1) (Integer comparison)

Note: Stack() does create a stack object if abstractmethod is NOT used.

Stack Arrays:

Stacks implemented with arrays often use 2 elements:

- An array to store the items in order of which they arrive:
  - Length of stack indicates max capacity for the stack
- An integer indicating the first empty space in the array
  - We can reuse length, as the first empty space coincides with the stack's length

Invariants of such implementation:

- Valid data appears in the 0 ... length - 1 positions of the array (beyond that is "garbage")
- Element at top of stack is element in position length - 1

- Abstraction comes from ignoring implementation, not hiding it
- Constant variables should be in UPPERCASE

For stacks, if already full & a PUSH attempt is made,

- If array is empty and we try to pop:

  Raise an exception.

  Eg: `def pop(self) -> T:`

  `If self.is_empty:`
  `raise Exception("stack is empty")` ...

Peek also has similar implementation to Pop.

Use of Array Stack:

ie. getting a string and reversing the order

- Take string and create a ADT String for each char
- Pop each character and concatenate it to a new string

Set ADT:

- A set ADT is used to store items.
  - An item is called a member (or an element) of a set
- Items in a set are unordered
  - But access to them should be fast!
  - And no duplicates are allowed
- Items can be added or removed & sets can be compared, merged, etc.
- Infinite set example: $\mathbb{R}$, $\mathbb{N}$, $\mathbb{Z}$, etc

Set Operations:

- Similar to mathematics with regards to arithmetic operations (+, -, ∪, ∩)
- Sets CANNOT have duplicates

Sets Invariant: Valid data appears from 0 to size - 1

ArraySet:

- Import the parent ArraySet class
- Have Min_Capacity = 1 in constructor
- Use methods from the parent class (implement the abstract methods)
- Remember, for "Set Empty" or "Set Full", use boolean logic
- Full capacity: return len(self) == len(self.array) → Elements given = size of array-set

Big O: set.__init__ calls clear
clear is O(1)
BUT, it also depends on ArrayR which is O(capacity) since Min_Capacity is const. O(1)
∴ Big O is O(capacity).

Implementing __contains__:

- This method implements the member operation

```
def __contains__(self, item: T) -> bool:
    for i in range(self.size):          <-- Iterates N times, N = size of set
        if item == self.array[i]:       <-- This costs O(comp) & depends on the type of
            return True                 --> O(1) | items we are comparing
    return False --> O(1)
```

∴ Overall cost: O(N · comp).

Implementing add :

```
def add(self, item: T) -> None:
    if item not in self:                    → O(N. comp)
        if self.is_full():                  → O(1) - prev. implementation [is_full]
            raise Exception("Set is full")

    self.array[self.size] = item            → O(1)
    self.size += 1
```

∴ Big O: O(N. Comp)

Implementing Remove :

```
def remove(self, item: T) -> None:

    for i in range(self.size):              ← N times
        if item == self.array[i]:           ← O(comp) - complexity of item comparison
            self.array[i] = self.array[self.size - 1]
            self.size -= 1                  → O(1)
            break
    else:
        raise KeyError(item)
```

∴ Big O = O(N. comp)

If you need to check size of set more often, use Arrays

Implementing Union:

```
def union(self, other: ArraySet[T]) -> ASet[T]:
    res = ArraySet(len(self.array) + len(other.array))   --> O(N+M)

    for i in range(len(self)):
        res.array[i] = self.array[i]

    for j in range(len(other)):   <-- Iterating over M elements other
        res.add(other.array[j])       and adding each to res.

    return res
```

∴ Big O: Complexity of add is O(size)
Size is comp x (N+M) in the worst case

∴ O(M · comp(N+M))

Set ADT with Bit Vectors:

Bit Vectors:

- Let items be positive integers
- Represent our set using a vector of bits
  - Item i belongs to the set if the vector has value 1 in the position i - 1
  - Otherwise, item is not in the set!
- We will use bitwise operations

Remember: Sets are unordered

Python magic methods: - Start and end with double underscores
- Not meant to be called directly (Python does it behind the scenes)
- Aim is to overload operations

- Bit Vectors: In BV, you need n elements to represent at most N elements
- Our computers use bit-wise logic:

A	B	AND A & B	OR A | B	AND NOT A & ~B
0	0	0	0	0
0	1	0	1	0
1	0	0	1	1
1	1	1	1	0

bitVector | (1 << p) ⟶ Adds an element to the list

If you do 1<<3, you move element 1, 3 units to the left

Question Bank Practice Applied 1:

1.
- The main idea of BubbleSort is to sort a list by taking an element and comparing that element to the one next to it (left) and swapping the element if it is bigger and moving on.
- The largest no. of swaps that can happen with Bubble Sort are $n-1$
- Best Case Time Complexity : $O(n)$
  Worst Case Time Complexity : $O(n^2)$
- The best case time complexity is $O(n)$ because this assumes the list is sorted and hence goes through the list of input size n just to verify if it is sorted. The worst case time complexity is $O(n^2)$ because the comparison goes $n(n-1)$ elements which gives as n^2-n but since we only care about the dominant term, we can say it is $O(n^2)$

2. [5, 20, -5, 10, 3] after one iteration of Bubble Sort becomes
[5, -5, 10, 3, 20]

[-5, -20, -5, -10, 3] after one iteration of Bubble Sort becomes
[-20, -10, -5, -5, 3]

3. Bubble Sort only stops when Check!
An optimisation of this could be adding a boolean expression to "swapped", where if after one iteration "not swapped" outputs True, then the algorithm halts meaning the list is sorted.

4. A sorting algorithm is "incremental" if we can add another element to the list and it will still sort it with one or more iterations. Bubble Sort is incremental if the added element still ensures relative order

5. A sorting algorithm is said to be stable if it maintains relative order. ie. if we have an input of 4,3,5,7,9 and we have another list 8,3,5,7,9, if 3 is before 5 on both lists, then it is stable. Bubble sort is stable.

6. Selection sort compares the list and keeps storing the smallest element of the list. It looks through the list if it finds a smaller element than the current smallest element, then it swaps it to be in the 0^{th} index and keeps repeating the process. If the smallest element is in the 0^{th} index then it takes the list next to it as the next point of comparison and so on.
Best and Worst Case time complexity for Selection Sort is $O(n^2)$ as it always has to traverse the list to find the smallest element & swap it no matter whether it is sorted or in reverse order.

7. Selection sort after one iteration: $[5, 20, -5, 10, 3] \rightarrow [-5, 20, 5, 10, 3]$
Selection sort after one iteration: $[6, 5, 4, 3, 2, 1] \rightarrow [1, 5, 4, 3, 2, 6]$

8. Selection Sort is not stable as we would not know the relative order as it only takes the minimum element each time.

9. Selection Sort is not incremental as the entire algorithm would have to be repeated if a new element is added.

10. Insertion sort takes an element, separates it into a list that is "sorted" and takes the next element in the "unsorted" list and puts it into the correct placeholder of the sorted list.
Best and Worst Case complexity of insertion sort is $O(n)$ & $O(n^2)$ respectively as it has to do n comparisons & n shifts & for best case only n comparisons and no shift as the list would already be in order.

11. Insertion Sort after one iteration : [5, 20, -5, 10, 3] ⟶ [5, 20, -5, 10, 3]
Insertion Sort after one iteration : [-7, -1, -4, 4, 5, 6] ⟶ [-7, -1, -4, 4, 5, 6]

12. Insertion Sort is irreversible thanks to the marker as the list will already assumed to be sorted meaning the new element can be inserted into its right place into the sorted list straight away with only one step.

13. Insertion Sort is stable as the sorted list would maintain relative order

Algorithms & Complexity:

1. Check!

2. If the algorithm is NOT correctly implemented to halt, then it can keep going infinitely.

3. Check!

4. $f(n) = O(g(n))$ means that $g(n)$ must be a function within $O(n)$ meaning $g(n)$ has to be the asymptotical upper-bound for $g(n)$ & that there has to be a constant c, n such that $n \geq n_0$.

5. a) $f(n)$: $\sqrt{n}$ as logarithmic is slower
b) $g(n)$: n^2 as quadratic grows faster than logarithmic
c) $f(n)$: 2^n as exponential grows faster than quadratic

7. a)
b) Check!

9. O(n) as constant is negligible in the large output scale which is what we need to consider for time complexity.

10. O(1) as the element size doesn't matter since function is set to iterate a constant time regardless

11. Same reason as above but instead of constant we have n^2 iterations meaning we will have a complexity of $O(n^2)$.

12. O(n) as though function is constant, we want complexity in terms of input size

Check !

13. Check!

14. Constant Time Complexity is denoted as O(1).

15. $O(n^2)$

17. $f_1(n) \cdot f_2(n) = O(n^2) \cdot O(n \log n) = O(n^3 \log n)$

19. Best & Worst case time complexity is the min. & max. time an algorithm would take to solve a problem based on an input size of n.

20. Should be analysed with respect to input size.

Worst case complexity 1 : $O(\log n)$
" " " 2 : $O(\log n)$
" " " 3 : $O(n^2)$

Best case complexity 1 : $O(\log n)$
" " " 2 : $O(\log n)$
" " " 3 : $O(n^2)$

Queue ADT:

- Queue ADT follows a first in first out method [FIFO]
- Like stacks, access to other elements is restricted.
- VERY similar implementation to Stack ADT.

Linear Queue:

- Add items at the rear [append]
- Take items from the front [serve]

Invariant: Valid data appears from front to rear - 1, and rear - front equals length

Linear Queue Implementation:

Big O: Queue.__init__ : O(1)
Clear : O(1)
Self.array: O(MAX CAPACITY) since min capacity is const. & max is O(1)

Implementing append:

- Raise exception if list is already full
- Then add item to rear & raise rear by 1

Implementing Serve:

- If queue is already empty, raise an exception
- Reduce length by one, make item at front of list & increase front by 1.

Implementing is_full:

- If len(self) == len(self.array)
- Whenever rear is pointing out
- There is not more space left!

If rear is full, front empty cells will be wasted.

Circular Queues:

- Simulated by allowing rear & front to wrap around eachother
- Similair implementation to Linear Queues
- len(self) == len(self.array) ⇒ List is full
- Time complexity: O(1)

Invariant: Valid data appears from:
If front <= rear: from front to rear-1, in that order
Else: from front to the end of the array & from 0 to rear-1, in that order

Extending and Using our Queue:

Extending the class to print elements from front to rear
↳ a method within the CircularQueue ADT.
↳ Implementation on Mac folder

Big O: Loop ALWAYS executed len(self) times. ∴ Best = Worst.
Inside loop, everything BUT print is fixed
∴ len(self) × (1×m) ≃ len(self)×m

BEST = WORST: O(len(self)·m)

Greater func:

Returns bool [True] iff q_2 is at least as long as queue 1 AND every element in Queue 1 is less or equal than the element in the same position in Queue 2.

You are allowed to modify the queues.

Implementation on MAC.

Common Queue Application: Printers, keyboards, etc.

List ADT:

- Elements have an order (similar to queues & stacks)
- Must have direct access to first element (head)
- From one position, you can always access the "next"
- Set of operations include: – Creating, accessing, computing the length
  – Testing whether the list is empty (not for full)
  – Adding, deleting, finding, retrieving & modifying elements.

Note: Lists are mutable, Tuples are NOT.

What can we do:

– Create a list, get/set the element at given element, compute the length
- Determine whether is empty
- Find position or index of an item (if in list)
– Insert an item at Position P
- Append an item
- Remove first occurence of an item

PTD!

- Delete & return the item in Position P [Python calls this Pop]
- Clear the list.

Implementation on MAC 1008 Folder

Lists Implemented with Arrays:

Implementation on MAC

Time Complexity: Everything is Constant BUT ArrayR.
∴ O(max_capacity)
AND
Best Case = Worst Case.

Invariant: Valid elements btwn 0 and self.length - 1, the head is at 0.

- Linear, sequential search in indices of a list to find the element.

Note: Raising Errors/Exceptions is NOT a part of Big O.

BEST CASE ≠ WORST CASE.

Best Case: Item is Start of list
∴ Const. + n + Cons. → O(n) or O(comp==)

Worst Case: Item is end of list.

len(self) × (Constant tm) + Constant → O(len(self) xm) - OR O(len(self) x comp==)

Big O for delete_at_index:

O(len(self)) → Worst, O(1) → Best case

Implementation on MAC

Implementing append:

- Implementation on MAC.
- No is_full method for lists
- If array is not full, just add the new element into a empty cell.
- If current array is full, create a new array with ie. double capacity, copy elements into new array with their same respective indices & use THIS as your array, then you can continue to append new elements.

Big O: Best Case: List not full: O(1)
Worst Case: O(len(self)) due to list being full

Note: _newsize should have O(1)

Insert method:

- Similar to append BUT adds the element at the specified index rather than the end of the list.

4. (i) Main properties of ListADT is access to other elements
(ii)
(iii)

Week 2 Question Bank...

1. An Abstract Data Type (ADT) specifies a set of operations that can be performed on data & the behaviour of its operations. It differs from a data type because data type specifies the kind of data that is stored & the basic operations whilst ADT gives more freedom to the implementation. Examples of ADT's are Stacks & examples of DataTypes in Python are Integers.

2. The key idea behind ADT's are giving the freedom to its implementation later on in the code when it is actually needed. It only gives the rules behind the data type whilst leaving its implementation free to the user's choice.

3. Capacity refers to the max. elements an ADT implemented with an Array can hold whilst length is how many of those elements are currently in it.

4. - The main property of Stack ADT is FIFO.
   - Key operations are push, pop, peep, is_empty, is_full, getItem
   - A few applications of Stack the undo mechanism

5. Best Case & Worst Case for push() and pop() O(1) because in both cases, the input size is always only one item and whether adding, removing [Best case] or raising an "it full" or an "is empty" exception [worst case] will not have an impact as it will always be constant as only the top element is the one we have to deal with.

6. - Set ADT can be implemented also using an Array that is Sorted
   - Main properties are it is unordered, no duplicates allowed and allows a method to see if an element is part of the set.
   - Add, Remove, contains, Union, Intersection, Size, difference

7. - A bit vector can represent a number of elements using binary values "1" or "0" where "1" represents an element that is present.
- Elements that can be represented using BV's are integers
- Bit Vectors operate by presenting 1 for an element present & 0 for the opp?
- Example: Bit Index: 0, 1, 2, 3, 4, 5
  101010 means the set contains {0, 2, 4}.

8. Cons of Bit Vector are smaller complexity for most basic operations
Cons of " " " fixed size for the bits it can hold, long complexity for calculating the length.
Cons of Set Array ADT (first) Con is opposite of bit vectors.

9. Array Set ADT Complexities:

add (): O(1)*, O(n)	__len__ (): O(1)**, O(1)	intersection(): O(n) O(n*m)
remove (): O(1), O(n)*	union (): O(n)***, O(n+m)	

*. O(1), can be added directly, O(n), if array needs to be resized.
**. A counter that is updated seperately with each method to always update
***. O(n), small set minimal overlap. O(n+m), larger set, scan all elements of both sets for overlaps.
*****. O(n) small overlap, O(n×m) check each element of one set w every other element of next set to see if they match.

10. BV Set ADT Complexities:

add (): O(1)*, O(1)	__len__ (): O(elems), O(elems)	intersection(): O(1)*, O(1)
remove (): O(1), O(1)*	union (): O(1)*, O(1)	

*. All can be done with simple bitwise operators since BV sets only deal with integers.
**. As regardless of size, has to go through all the elements in the set anyway to manually "count".

Sorted List ADT:

- A list ADT where all elements are sorted (in increasing order for FITTOOS)
  ↓
  An additional invariant of this class
- Looks very similar to List ADT but without the set item, append or insert

Methods:

Index — Linear search works!

Invariant: every item is greater than or equal than the previous one

Implementation of Index on Mac 1000 folder.

Better than List ADT because it is more efficient though time complexity is same

Binary Search:

- Linear search for sorted list is **only better** when the element is **not found**
- **Binary search** is smth that is allways better
- The "middle search":

```
If (value == middle element)
        value is found
elif (value < middle element)
        Search left-half of list with the same method
else
        Search right-half of list with same method
                Keep halving until element is found.
```

Linked Lists:

Arrays are fixed size, data items stored sequentially, each item occupies some amount of space

Resizing in Arrays is difficult and expensive

So we have:

Linked Nodes: Each node contains one or more data items & one or more links to other nodes

To do this, we create a class with 2 instance variables

Linked Node Data Structures Give:

- fast deletion of an item (no need of reshuffling)
- fast addition of an item (create a node, insert it (link it) into the correct position)
- Easily resizeable: create / delete a node, it is only full if there is no memory left to create a new node.
- Less memory used than an array if the array is relatively empty.

Linked Node Data Structures (→)ve:

- More memory used than an array if the array is relatively full
- for every item, the node uses twice the space. So if array is more than half full, then it takes less memory than link nodes.

Instance Variables:

- Only need one component: A reference to the head node
- Since our base class also created the length, this can be used too.

- From the head, we can access every other node

Invariant for __init__: the number of valid elements is self.length & are found via the node links starting from the head

delete_at_index: Link the previous element to the element after the element that is being deleted and return the element being deleted.

Insert method:
- Create a new node with the item
- Find the node previous to the index
- Set the link of the new node to the link of the previous
- Set the link of previous to the new node
- If want to insert at index 0, then change head.

Implementing Index. To find the actual index:
- A linear search until we find the item or we reach the end of the list
- That keeps an index to count the number of times we move via the links.
- Start at index 0 and point to the head.
- To stop the search, modify it to have item < current.item, but, Binary Search is not possible! In linked nodes, we can only go one direction. If we do the middle element method, then we cannot go left, we only have to go right.

Week 3 Theoretical Question Bank:

1. (i) The main property of a Queue ADT is that it follows FIFO and access to other elements is restricted.

(ii) Key operations of the Queue ADT are length, is_full, append and serve.

(iii) A few applications of Queues is online concert ticket purchases, waiting on hold for a telephone related service.

2. A linear queue has only one use for its space meaning that once element is removed that index cannot be used. But this can be done in circular queues as we can use the modulus function to wrap around (rear to front). and reuse elements even after they have been deleted.
Another difference is that linear queue can become full but with circular queues, once rear points to front, we can reuse indexes over & over again.
Circular ques are better overall as they make efficient use of memory due to their wrap around feature.

3.
Append() Best Case: O(1)
" Worst " : O(1)
Serve() Best " : O(1)
" Worst " : O(1)
is_full() Best " : O(1)
" Worst " : O(1)

} Constant time complexity because it is similar to linear queue. All the instructions are constant time including modulus function so the is also constant time. For worst case, it is same too.

4. (i) Main property of a List ADT is access to other elements and their sorted order.

(ii) Add, Remove, is_empty and is_full

(iii) UNO Cards, Shopping lists.

5. insert()
index()
delete_at_index()

} Check with page 26.

While player.hand != 0:

current_player = 0

if self.current_player.card color = self.card color
(if card NOT crazy:)
OR
= self.card label:

play card.

if card not [0, 9]:

play crazy. → next player. next player

elif card = Reverse no. :

reverse

elif card = skip no. :

skip

next_player.

Continue playing :

if player's last card = draw

next player

draw card [2 OR 4]

end.

```
{ "Student accounts": [
  { "name": "ABC",
    "Age": "20",
    "Sex": "F"
  },
  {
    "Sex": "M",
    "name": "xyz",
    "Age": "25"
  }
] }
```

Request body in Postman

Insert into Student (name, age, sex) values (

let name, sex, age

if card label & color not match:

self. draw pile

→ next player

elif card is crazy:

→ if card = draw 2 no.

if card not [0, 9]:

play crazy. → next player. next player

elif Reverse no

Take 2 cards Next player

Move to current_player +2

If draw pile.isempty():

Temp: self.discard pile.pop()

temp: Store top.

for all discard:

Store in array. (create array)

Shuffle. array

add draw pile [stack]

Temp card = new discard pile stack 1st card.

Hash Tables:

Container ADTs: Store and remove items independent of contents.

eg: List ADT, Stack ADT, Queue ADT

Core operations: Add, Delete, Search (for lists)

- Dictionary ADT is also another kind of container type to store objects

- Main Difference: - Objects are uniquely identified by a label or key
  - Defined in Python using curly brackets or calling dict()
  - Accessed with the usual square brackets (as lists)

Operations of Dict ADT: Search, Add, Delete, Update

- Remember: - Keys are unique so you can update them if they exist or add a key if they do.
  - If you search for a key that doesn't exist, you will get a KeyError.

- In python, dictionaries are implemented using Hash Tables.

Reason for Hash Tables:

- Assume we want to store a very significant amount of data (a big N)
- Assume we will need to perform the following operations relatively often:
  - Search for an item
  - Add a new item (or update its data
  - Delete an item (optional)

- But we do NOT need to traverse them in a particular order or sort them (at least not often)
  - Traversing all is fine as long as the order is irrelevant
  - If you often need to traverse in a particular order, use a different ADT.

Data Types for Hash Tables:

Stacks (LIFO): Not suitable for searching/deleting

Queues (FIFO): " " " " "

Unsorted Lists (N = len(list)): Search: O(N x Compeq) worst in linked list & array

Add: O(1) worst in linked list (1st element) & arrays (last element)

Delete: O(N x Compeq) worst in linked lists and arrays

- Sorted Lists (N = len(list)):
  - Search: O(N * Compeq) worst in linked lists; O(log N) in array
  - Add: O(N * Compeq) worst in linked lists & arrays
  - Delete: O(N * Compeq) worst in linked lists & arrays.

Aim of Hash Tables:

- Have Constant time operation O(1)
- Worst case can still be O(N) if not careful – need to construct the hash table well

How can we do this?

- Use arrays: Constant time access to a given position

  BUT each item MUST have an assigned position.

Hash Tables Data Type:

Data: - Items to be stored

- Each item must have a unique key

Data Structure to implement Hash Table:

- Large array (also reffered to as the Hash Table)

Basic Operations:

- Hash function: maps a unique key to an array position
- Add, Search, Delete

Hash Function:

- Maps from a set of keys K to a set of hash values H.

Key → [Hash fnc.] → Hash Value

- Hash function's properties:

Basic Properties: - Type dependent: - depends on the type of the item's key
- Return value within array's range (0... TABLESIZE - 1)

- Desirable: - Fast, a slow hash fnc. will degrade performance
- So, should not have too many arithmetic operations

- Minimise Collisions (Two keys mapped to same hash value)
- Distribute keys by hash values uniformly
- Special case: maps every key into a different hash (perfect hashing)

Almost impossible IRL

- Perfect Hash fnc's are RARE:
  - Rely on very particular properties of keys
  - Almost impossible to get in practice

Universal hashing: (Nice in theory):
- Applies a family of hash fnc's F, such that |F| = k
- Given a key, picks a hash function from F at random
- Guarantees the chance of a collision to be ≤ k/Tablesize

* Good functions aim to get closer to universal hashing

How to define Hash Fnc's:

- If key is an integer randomly distributed:
  - Position = key % TableSize (Random & Fast)

If not: If you have a n-bit key, (eg. on MAC FITBOOK week 5 Folder)
Modify the key until all bits count
- Checksum should not be considered & Category codes should not be changed.

- More elements in the key you use, better the hash function

- Using all elements is not enough to guarantee a good spread of possible hashes
- Need smth that uses all elements & takes into account its position in the key

- We want smth in range of our TABLESIZE.
  - Possible solution: If no. too big: use % TABLESIZE
    If no. too small: Convert 0...1 and x TABLESIZE
    but be careful to not overflow or underflow
  - Possible sol: mod/multiply at every step

Horner's Method:

- $h = ((\dots(a_0x + a_1)x + \dots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$
- Mods at each step, thus casting out multiples of TABLESIZE

If key is not random, use a prime table size

- If multiplying by another factor & modding, make sure they are relatively prime / coprime.

- Having common factors is likely to result in keys with close hash values (clustering)

- Even with the same values (collisions)

- Choose coefficients in a pseudo-random fashion
- Use a different coefficient for each key position

} More effective

Note: By ensuring that all coefficients & Table size is prime, we produce a Hash Function that is fast and produces sparse Hash values.

Collision:

- To handle collisions:

  - Use Open addressing (multiple schemes exist)
  - Seperate Chaining: - Simple to implement
    - Apply a linked list (or a balanced tree) to represent colliding keys:
      (store more than one item, relatively easy to insert/update).

      Examples on Week 5 FITIOOS folder on MAC

Conflict Resolution (Linear Probing):

- Hash Table Operations: Add
  - Apply the hash fnc. to get a hash value (position) N.
  - Try to add key at position N
  - Deal with collision/conflict if any.

NOTE: hash(key1) = hash(key2) is a collision
Conflict is assumed to occur when a position in the hash table we attempt to use for a given key is already occupied

Conflict Resolution:

- Seperate Chaining: A linked list / tree for each position in an array
- Open Addressing: · Each array position contains a single item
  · Upon collision, either update (same key) or use an empty space to store the new item (which empty space depends on technique)
  · Requires an array double the size of no. of elems.
  · Thus, must estimate no. of elems in advance or risk dynamic resizing
- Linear Probing: · If array[N] is empty: put item there
  · If there is already an item there with:
    - A different key: · Search for first empty space in array from N+1
      · Add item there (if any)
    - Same key: update the data associated with the key.

→ Linear search from N until empty slot is found.

Note: Must deal with: · full table (to avoid ∞ loop)
· Restarting from position 0 if end of table is reached

Note: When adding single key to a hash table, you can only have a max. of one collision, everything else is known as Probing.

- When table size is > 50%, conflicts and collisions increase, efficiency decreases.

- If we want to store key & data for keys, use a Tuple, (key, data). But, remember, Tuples are Read Only.

- Algo for add(key, data) for linear Probing:
  - Get position N using hash fnc., N = hash(key)
  - If array[N] is empty, put Item(key, data) there.
  - Else, if already an Item there:
    - If keys are same: update data.
    - If different key, keep looking in next cell (wrap around)
      - If never found empty spot, rehash: create a bigger array & re-insert all items
    - We MUST traverse table before we can rehash
      (just in case key exists in table & we update it)

Conflict Resolution Search:

- Search for an item with hash value N:
  - Perform a linear search from array[N] until either the item or an empty space is found
    (if so, raise a KeyError(key) exception)
    But remember again to deal with full table and restart with 0 if end of table is reached

- Algo for search(key, data):
  - Get position N using the hash fnc., N = hash(key)
  - If array[N] is empty, raise a key error(key) exception
  - Else if an item is already there: - If item has same key: return associated data
    - If item has different key, keep looking
      If no key & no empty spot, raise KeyError(key) exception.

Note: Use __setitem__ for adding, __getitem__ for searching

Deleting in Linear Probing:

- Use search fnc. to find item
- If found at N_y should we delete & leave it empty?
  No as empty spots have meaning in Linear Probing.
  Invariant in Linear Probing: If an item with hash(key) = N is in the table, it will always appear btwn N & the first empty position (wrapping around)
- Even shuffling back one position will NOT guarantee correct positions.

- One Solution: If found at N_y re insert every item from N+1 to the first empty position
  BUT time consuming (if we have large data sets)

- Another Solution: - Use a special symbol (a sentinal) to denote delete.
  - Modify add & search to take symbol into account
    for search: treat sentinal as you would treat a cell with a different key to the one you are searching for (keep on looking)
    - If found, move it to the first deleted cell you found.
    for add: treat it as empty & add it in the first deleted cell

Load Factor: total number of items / Table size (how full hash table is)
Cluster: sequence of full hash table slots (ie. without empty slots) (how many positions we have before empty spot)
Cluster can form even when the load is small
Once a Cluster forms, it tends to grow larger
- Items that get added to a value within a cluster, gets added at the end making it bigger
- This might involve more than one hash value.

Tendency for Clusters to occur when load > 0.5

Low speed on Clustering:

- Adding a key with hash value N can drastically increase the search time for keys with with values other than N.
- Deletion can also be time consuming as entire cluster needs to be re-hashed
- Start to under-deliver on O(1) promise
- If implemented with arrays, can become quickly full => resize => expensive
- To resolve: Reduce clustering by taking bigger and bigger steps (rather than always using +1).
- Try to keep load factor under 0.5 (smaller linear probe chains)

Linked Lists and Iterators:

- Linked Storage: Unknown list size (no need to resize by copying)
  - Memory is an issue, not time
  Many insertions and deletions needed within (rather than at the end)
  - If one does not have direct access to positions though
  Most operations need traversal of the list from the first element

- Arrays (Contiguous Storage): - Known no. of elements (no need for links)
  - Few insertions & deletions needed within (no shuffling)
  - Lots of searches on a sorted list (can use binary search ⇒ faster)
  - Access to elements by their positions (Constant time)

We need Iterators to create duplicates / modify the class without actually accessing the internals of the class (ie self.link should not be accessed)

Iterators:

- Is an object other than the list because:
  - It needs to change (move through list) without changing the list.
  - We might need several iterators on a list.

- In python, iterators objects are returned by the method __iter__
  - It's __init__ performs the "beggining-of-iteration" initialisation.
  - To return the next element, we use the __next__ method in Python

Iterators iterate through an object without actually modifying the object.

When iterating with linked list, linked list must also have a iterator function in it.

Using Iterators:

- Iterators _next_ method has a constant, O(1), time complexity.
- Common operations on an iterator's output include:
  - Perform some operation for every element (eg. 3 * x)
  - Select a subset of elements that meet a condition (eg. x % 2 == 0)

- List Comprehensions allow you to do that AND return a list
- Generator expressions allow you to do that AND return an iterator

A Modifying Iterator:

- With current understanding, iterator cannot modify the objects.
- ∴ Modify the iterator object by adding:

  delete : returns the item pointed by current and deletes the node

  add : adds an item right before current

  has_next : returns True if there is a next item to be processed (ie. if current is not at the end of list)

  peek : returns the next item (the one pointed by current) without moving along.
  Raises StopIteration if there is no next

- Since we need to add and delete:
  - We are going to need not only a current but also a previous
  - Plus sometimes, we will need to change the head of the list.
    - We are going to need to keep the list itself

Linked Stacks:

- See FIT1008 Folder Week 5 for Push & Pop implementations

- Note: - Push() and Pop() in Linked Stacks have O(1) complexity.
  - We need to reference & return the item only, not the node for the method.

- Purpose of ADTs is for good code re-use

Pros of Linked Stack:
- Good to resize: Push is never full, so we can just add
- Pop uses less memory
  - Needs less space than array, if array is relatively empty (less than half)

Cons of Linked Stack:

- Needs more space (for links) than array, if array is relatively full
- A bit slower. Const. time but bigger const. as we need to create nodes as well

Linked Queues:

- With Linked Queues, we do not need circular queues as Linked Nodes do not have a fixed space and can thus, keep growing.

- Append:

With Linked Lists:

- Create a new node that contains item and points to None
- Link the current rear to it
- Make the new node the rear
- We need for the is_full check as linked nodes have no limit.

- Complexity for append with Linked Queues is O(1).

Serve method:

With Linked Queues, for serve:

- Identical to Array based implementation but move front along rather than increase it.
- Complexity: O(1)

- Application of Linked Queues: Check FIT1008 Week 5 Folder, Linked Queues sub folder

Examples on FIT1043 WEEK 7 Folder

In Decision Trees: Prediction is the most common value in each region
In Regression Trees: Prediction is usually the average value in each region

Recursion:

For all code implementation, look at FIT1008 Week 8 Folder.

- Solve a large problem by reducing it to one or more sub-problems that are of the same kind as the original and simpler to solve.

- Each of the sub-problems are to be solved using the same algorithm. This step has to be done UNTIL sub problems are so "simple" that they can be solved without further reduction (our base cases)

Candidate Problems for Recursion:

1. Must be possible to decompose them into simpler similar problems
2. At some point, the problems must become so simple that can be solved without further decomposition
3. Once all sub problems are solved, the solution to the original problem can be computed by combining these solutions.

- As a programmer, you need to know how to detect & solve the base cases, decompose problem into simpler subproblems that converge towards the base case, combine the sub-solutions together

Factorials:

- Determine the no. of permutations of a given no. of distinct elements.

- To program factorials: Assume $n \geq 0$ and factorial of $0 = 1$.

$$n! = (n-1)! \times n$$

↳ Can be done with a Recursive Method.
Base case is the result of 0! which is 1.

- For Recursion, must have at least one base case, at least one recursive call whose result is combined & convergence to base case.

Iteration vs. Recursion.

- With Iteration, you go start to finish but with recursion, you start with call based and go back to the start and run through for each call

- Iterations can be implemented using recursion if iterations are replaced by function calls or base case is the (negated) condition of the loop.
  - often needs an auxiliary function to prepare the converging arguments

- Recursion functions can be implemented using iteration by storing past results in either accumulators or a stack

Unary: A single recursive call (all previous code)
Binary: Two recursive calls ("find a route" example)
n-nary: n recursive calls

Direct vs. Indirect Recursion:

Direct: Recursive calls are calls to the same function

Indirect (or Mutual): Recursion through two or more methods (i.e. method p calls method q which in turn calls method p again.

Tail-Recursion:
- Where result of the recursive call is the result of the function
- That is: nothing is done in the "way back"
- Closest idea to Iteration
- Can be made into tail recursive by using an accumulator

- With Runtime Stacks, each recursive call reserves a new spot on the stack.

Gives Recursion:

- Sometimes more natural (i.e. Fibonacci)
- Easier to prove correct (due to mathematical foundations)
- Easier to analyse (due to mathematical foundations)

Gives Recursion:

- Run-time overhead (for tail-recursion, this will depend on quality of the compiler)
- Memory overhead (fewer local variables versus stack space for function call.