Lesson 2: Sets & Maps

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS | SPRING 2022

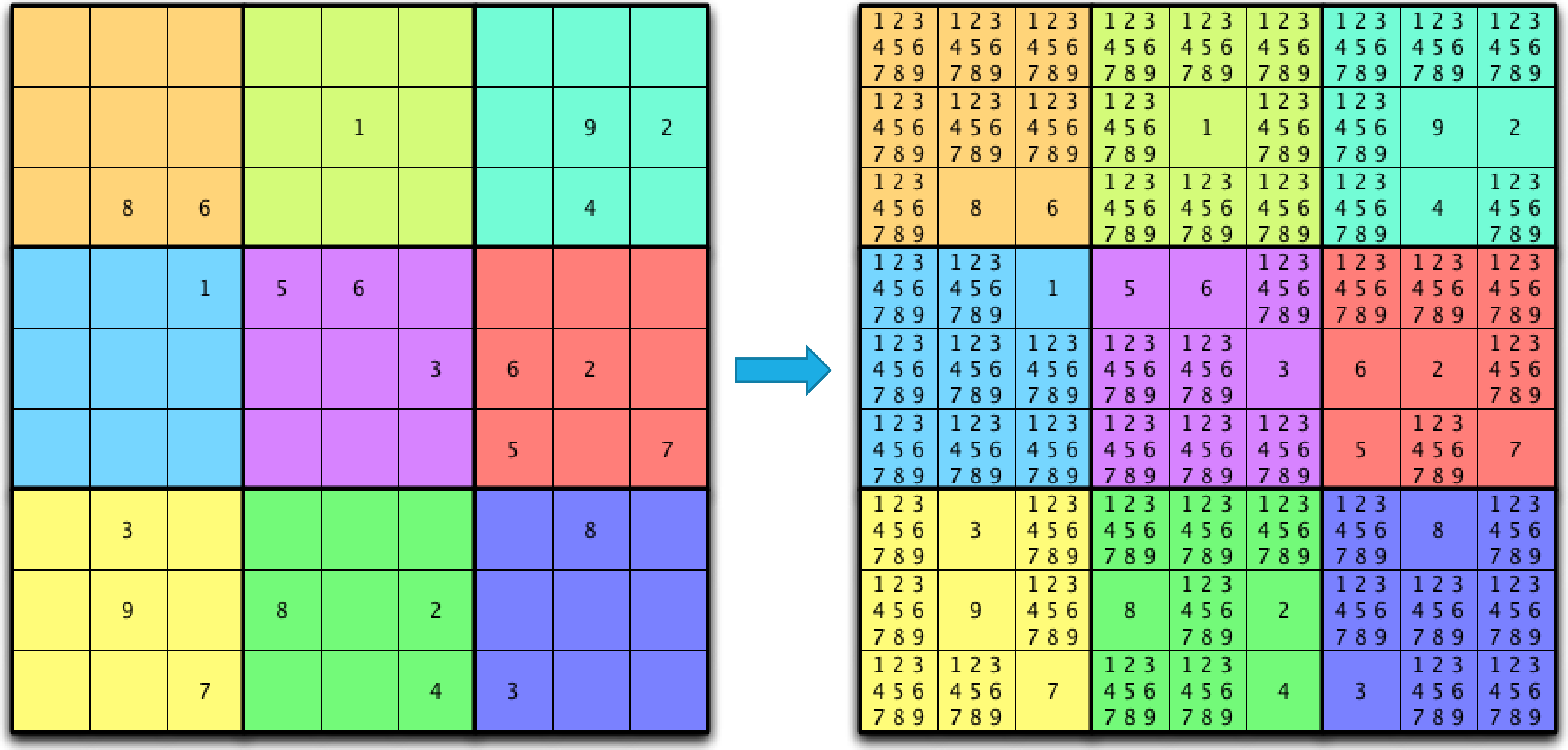
DR. ANDREY TIMOFEYEV

OUTLINE

- Introduction.
- Sets.
- Hashing.
- Collision resolution.
- Maps.
- Memoization.

INTRODUCTION (1)

- Consider a sudoku puzzle.



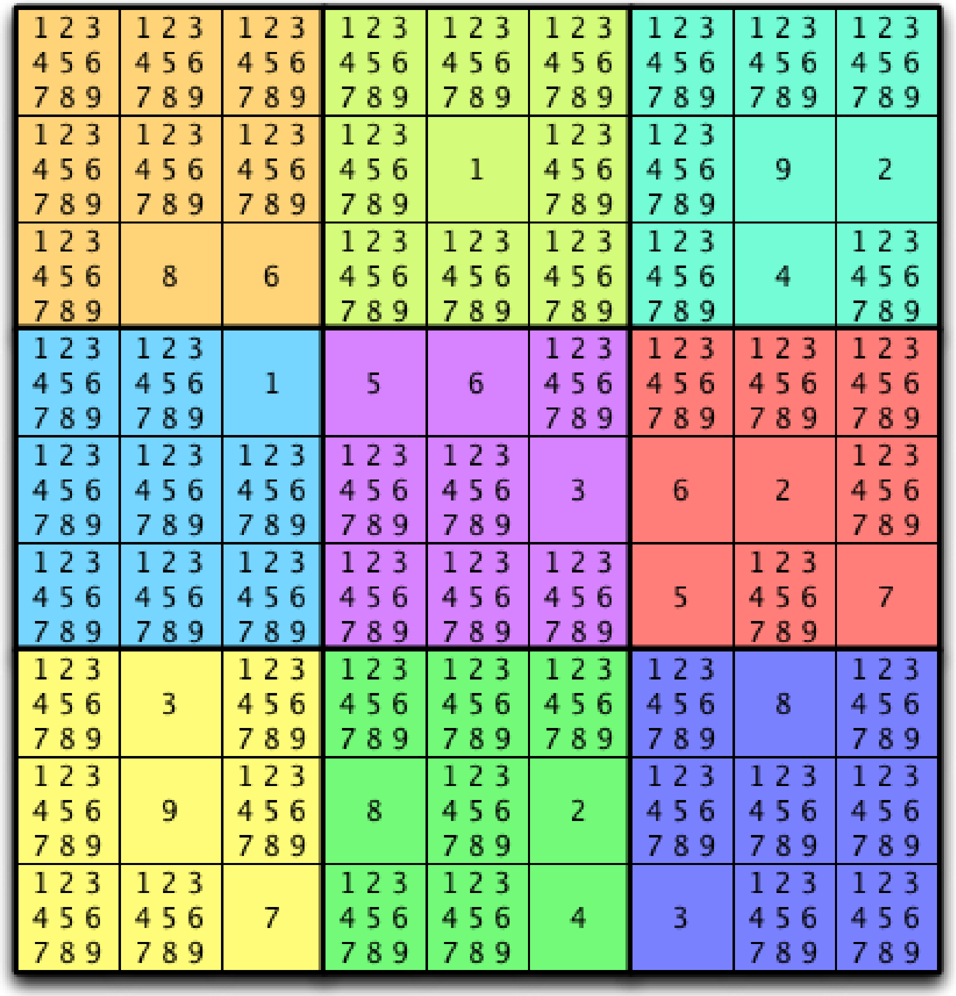
				1			9	2
	8	6					4	
		1	5	6				
					3	6	2	
						5		7
	3						8	
	9		8		2			
		7			4	3		

1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	2
1 2 3 4 5 6 7 8 9	8	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	5	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	3	6	2	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	7
1 2 3 4 5 6 7 8 9	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

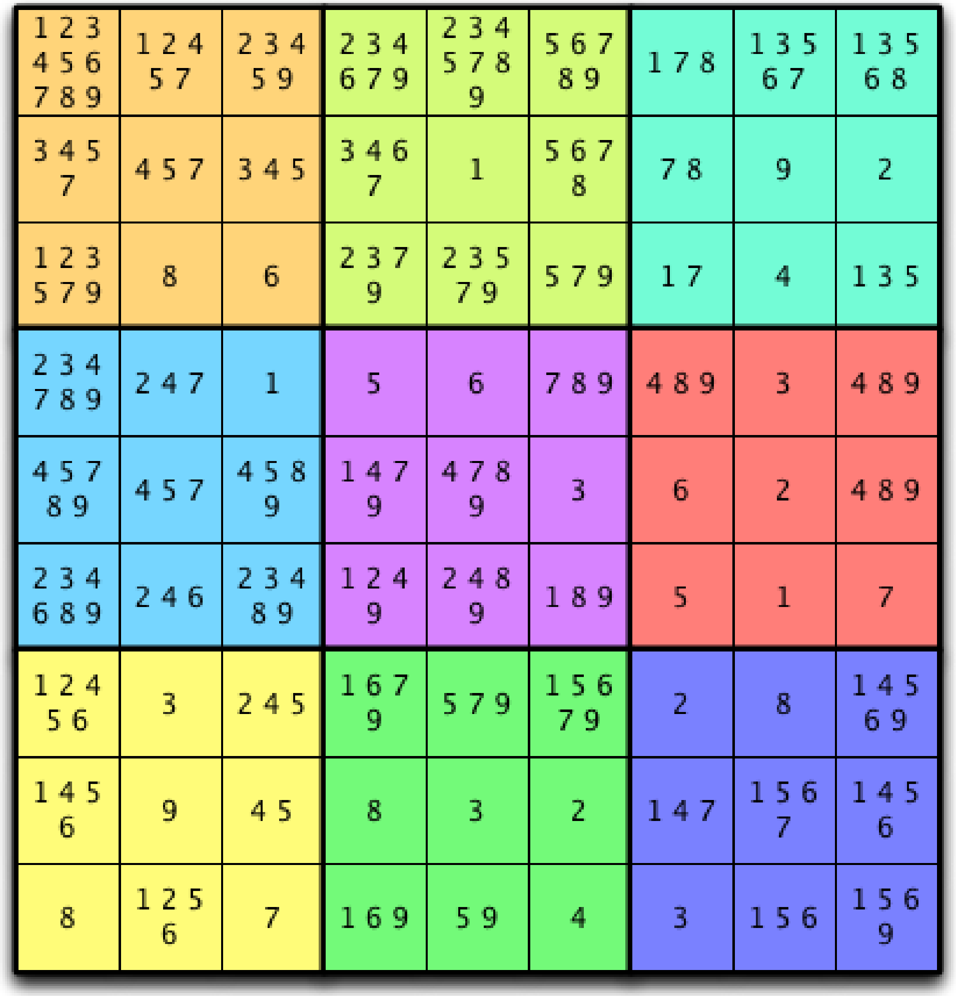
Sudoku puzzle process

INTRODUCTION (2)

- Consider a sudoku puzzle.



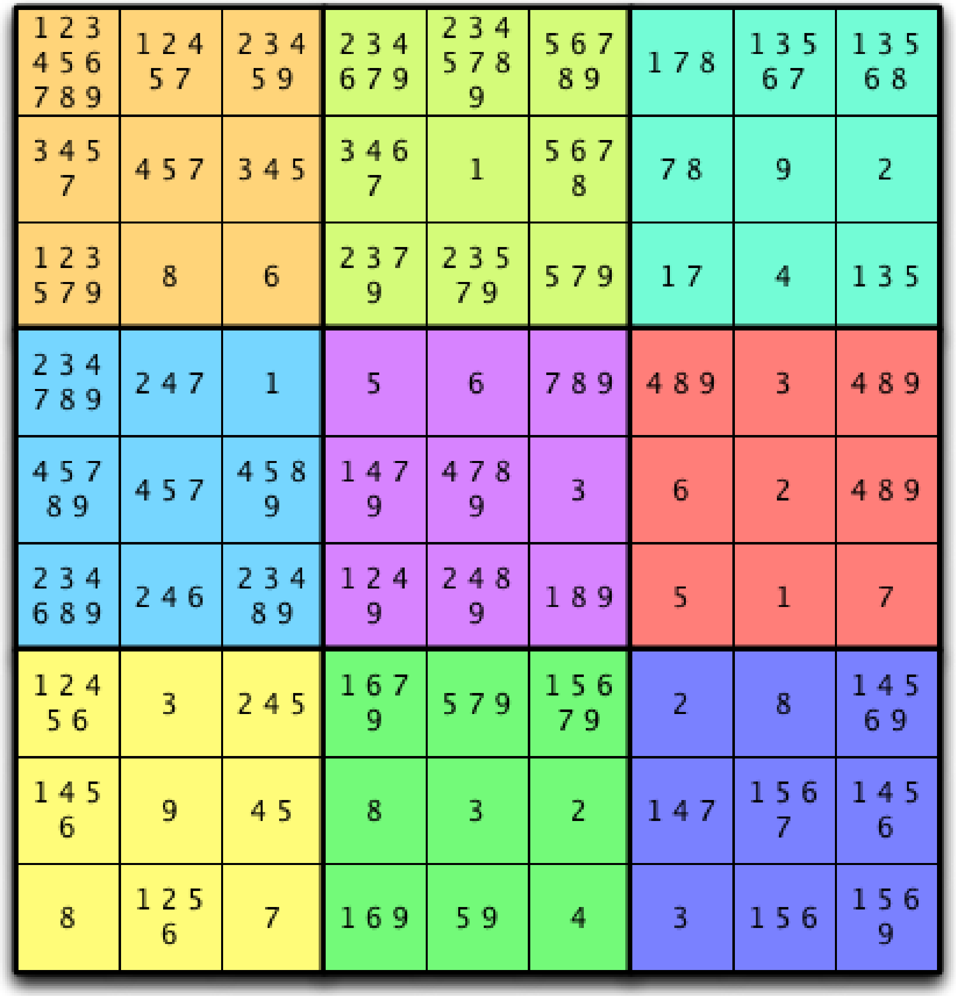
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	9	2
1 2 3 4 5 6 7 8 9	8	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1	5	6	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	3	6	2	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	5	1 2 3 4 5 6 7 8 9	7
1 2 3 4 5 6 7 8 9	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	9	1 2 3 4 5 6 7 8 9	8	1 2 3 4 5 6 7 8 9	2	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	4	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9



1 2 3 4 5 6 7 8 9	1 2 4 5 7	2 3 4 5 9	2 3 4 6 7 9	2 3 4 5 7 8 9	5 6 7 8 9	1 7 8	1 3 5 6 7	1 3 5 6 8
3 4 5 7	4 5 7	3 4 5	3 4 6 7	1	5 6 7 8	7 8	9	2
1 2 3 5 7 9	8	6	2 3 7 9	2 3 5 7 9	5 7 9	1 7	4	1 3 5
2 3 4 7 8 9	2 4 7	1	5	6	7 8 9	4 8 9	3	4 8 9
4 5 7 8 9	4 5 7	4 5 8 9	1 4 7 9	4 7 8 9	3	6	2	4 8 9
2 3 4 6 8 9	2 4 6	2 3 4 8 9	1 2 4 9	2 4 8 9	1 8 9	5	1	7
1 2 4 5 6	3	2 4 5	1 6 7 9	5 7 9	1 5 6 7 9	2	8	1 4 5 6 9
1 4 5 6	9	4 5	8	3	2	1 4 7	1 5 6 7	1 4 5 6
8	1 2 5 6	7	1 6 9	5 9	4	3	1 5 6	1 5 6 9

INTRODUCTION (3)

- **Two rules to solve Sudoku puzzles:**
  - Rule 1: Within a group look for cells that contain the same set of possible values. If the cardinality of the set matches the number of duplicate sets found, then the items of the duplicate sets may safely be removed from all non-duplicate sets in the group.
  - Rule 2: Look at each cell within a group and throw away all items that appear in other cells in the group. If left with only one value in the chosen cell, then it must appear in this cell and the cell may be updated by throwing away all other values that appear in the chosen cell.



1 2 3 4 5 6 7 8 9	1 2 4 5 7	2 3 4 5 9	2 3 4 6 7 9	2 3 4 5 7 8 9	5 6 7 8 9	1 7 8	1 3 5 6 7	1 3 5 6 8
3 4 5 7	4 5 7	3 4 5	3 4 6 7	1	5 6 7 8	7 8	9	2
1 2 3 5 7 9	8	6	2 3 7 9	2 3 5 7 9	5 7 9	1 7	4	1 3 5
2 3 4 7 8 9	2 4 7	1	5	6	7 8 9	4 8 9	3	4 8 9
4 5 7 8 9	4 5 7	4 5 8 9	1 4 7 9	4 7 8 9	3	6	2	4 8 9
2 3 4 6 8 9	2 4 6	2 3 4 8 9	1 2 4 9	2 4 8 9	1 8 9	5	1	7
1 2 4 5 6	3	2 4 5	1 6 7 9	5 7 9	1 5 6 7 9	2	8	1 4 5 6 9
1 4 5 6	9	4 5	8	3	2	1 4 7	1 5 6 7	1 4 5 6
8	1 2 5 6	7	1 6 9	5 9	4	3	1 5 6	1 5 6 9

Partially reduced Sudoku

SETS (1)

- **Sets** – collection of items that does not allow duplicates.
  - **Items**: integers, characters, strings, object.
- **Cardinality** of a set – number of items in the set.
- Python supports two types of sets:
  - **Set class.**
    - Object are mutable (can be changed after created).
  - **Frozenset class.**
    - Objects are immutable (cannot be changed after created).

SETS (2)

- Python **`set`** & **`frozenset`** operations reference.

Operation	Complexity	Syntax	Description
Set Creation	O(1)	**`s=set([iterable])`**	Calls the set constructor to create a set.
Set Creation	O(1)	**`s=frozenset([iterable])`**	Calls the frozenset constructor to create an frozenset object.
Cardinality	O(1)	**`len(s)`**	The number of elements in *s* is returned.
Membership	O(1)	**`e in s`**	Returns True if *e* is in *s* and False otherwise.
!Membership	O(1)	**`e not in s`**	Returns True if *e* is not in *s* and False otherwise.
Disjoint	O(n)	**`s.isdisjoint(t)`**	Returns True if *s* and *t* share no elements, and False otherwise.
Subset	O(n)	**`s.issubset(t)`**	Returns True if *s* is a subset of *t*, and False otherwise.
Superset	O(n)	**`s.issuperset(t)`**	Returns True if *s* is a superset of *t* and False otherwise.
Union	O(n)	**`s.union(t)`**	Returns a new set which contains all elements in *s* and *t*.
Intersection	O(n)	**`s.intersection(t)`**	Returns a new set which contains only the elements in both *s* and *t*.
Set Difference	O(n)	**`s.difference(t)`**	Returns a new set which contains the elements of *s* that are not in *t*.
Set Copy	O(n)	**`s.copy()`**	Returns a shallow copy of *s*.

Set & frozenset operations

SETS (3)

- Python **mutable `set`** operations reference.

Operation	Complexity	Syntax	Description
Union	O(n)	**`s.update(t)`**	Adds the contents of *t* to *s*.
Intersection	O(n)	**`s.intersection_update(t)`**	Updates s to contain only the intersection of the elements from *s* and *t*.
Set Difference	O(n)	**`s.difference_update(t)`**	Subtracts from *s* the elements of *t*.
Symmetric Difference	O(n)	**`s.symmetric_difference_update(t)`**	Updates *s* with the symmetric difference of *s* and *t*.
Add	O(1)	**`s.add(e)`**	Add the element *e* to the set *s*.
Remove	O(1)	**`s.remove(e)`**	Remove the element *e* from the set *s*. Raises *KeyError* if *e* does not exist in *s*.
Discard	O(1)	**`s.discard(e)`**	Remove the element *e* if it exists in *s* and ignore it otherwise.
Pop	O(1)	**`s.pop()`**	Remove an arbitrary element of *s*.
Clear	O(1)	**`s.clear()`**	Remove all the elements of *s* leaving the set empty.

Mutable set operations

HASHING (1)

- **Hashing** – process of **mapping** a **data** of arbitrary size to a fixed-sized **value**.
  - **Fixed-sized value** (aka **hash value**, **hash code**, **digest**, **hash**) can be used as an **index** in a **hash set**.
- **Hashing** allows performing **random access** in lists.
  - **Randomly accessible list** = any location within the list is **accessible** in **O(1)** time.
  - Achieved by using **item itself** to calculate the **index** where it will be stored.
  - **Index** is **calculated** by running the item through a **hashing function**.
- Python has a built-in `hash()` function and `__hash__` class method.
  - **Mutable** objects (e.g. lists) are not **hashable**.

HASHING (2)

- **Storing items.**
  - Item is stored in a list with an index calculated using hashing function.
    - Problem: list length << unique hash values.
    - Solution: `index = hash(item) % len(list)`
- **Collision resolution.**
  - Hash values are not unique, thus prone to collisions.
  - Problem: attempting to store items under same index.
  - Solution: collision avoidance techniques.
    - **Linear probing** – if collision then advance to the next empty location.
    - Double hashing, quadratic probing & separate chaining.
- **Load factor** (fullness of a hash set) = number of stored items / list length.
  - When **adding** items, if **load factor > 75%**, then **double** list size + **rehashing**.
  - When **removing** items, if **load factor < 25%**, then **halve** list size + **rehashing**.

Code example: Building HashSet class

Maps (1)

- **Maps** are used to **map** a set of **unique keys** to associated **values**.
  - *Map = dictionary = hash table = hash map.*
  - **Value** is looked up by a **reference** to a **key** in *O(1)* time.
    - **Hashing**.
- Many implementation **similarities** with **sets**.
  - Maps store **key-value** pairs when sets only store **items**.

Maps (2)

- Python **`dictionary`** operations reference.

Operation	Complexity	Syntax	Description
Dictionary Creation	O(1)	**`d = {[iterable]}`**	Calls the constructor to create a dictionary.
Size	O(1)	**`len(d)`**	The number of key/value pairs in the dictionary.
Membership	O(1)	**`k in d`**	Returns True if *k* is a key in *d,* False otherwise.
non-Membership	O(1)	**`k not in d`**	Returns True if *k* is not a key in *d,* False otherwise.
Add	O(1)	**`d[k] = v`**	Adds *(k,v)* as a key/value pair in *d*.
Lookup	O(1)	**`d[k]`**	Returns the value associated with the key, *k*. A *KeyError* exception is raised if *k* is not in *d*.
Lookup	O(1)	**`d.get(k[,default])`**	Returns *v* for the key/value pair *(k,v)*. If *k* is not in *d* returns *default* or *None* if not specified.
Remove Key/Value Pair	O(1)	**`del d[k]`**	Removes the *(k,v)* key value pair from *d*. Raises *KeyError* if *k* is not in *d*.
Items	O(1)	**`d.items()`**	Returns a view of the key/value pairs in *d*.
Keys	O(1)	**`d.keys()`**	Returns a view of the keys in *d*.
Values	O(1)	**`d.values()`**	Returns a view of the values in *d*.

Maps (3)

- Python **`dictionary`** operations reference (cont.)

Operation	Complexity	Syntax	Description
Pop	O(1)	**`d.pop(k)`**	Returns the value associated with key *k* and deletes the item. Raises *KeyError* if *k* is not in *d*.
Pop Item	O(1)	**`d.popitem()`**	Return an abritrary key/value pair, *(k,v)*, from *d*.
Set Default	O(1)	**`d.setdefault(k[,default])`**	Sets *k* as a key in *d* and maps *k* to *default* or *None* if not specified.
Update	O(n)	**`d.update(e)`**	Updates the dictionary, *d*, with the contents of dictionary *e*.
Clear	O(1)	**`d.clear()`**	Removes all key/value pairs from *d*.
Dictionary Copy	O(n)	**`d.copy()`**	Returns a shallow copy of *d*.

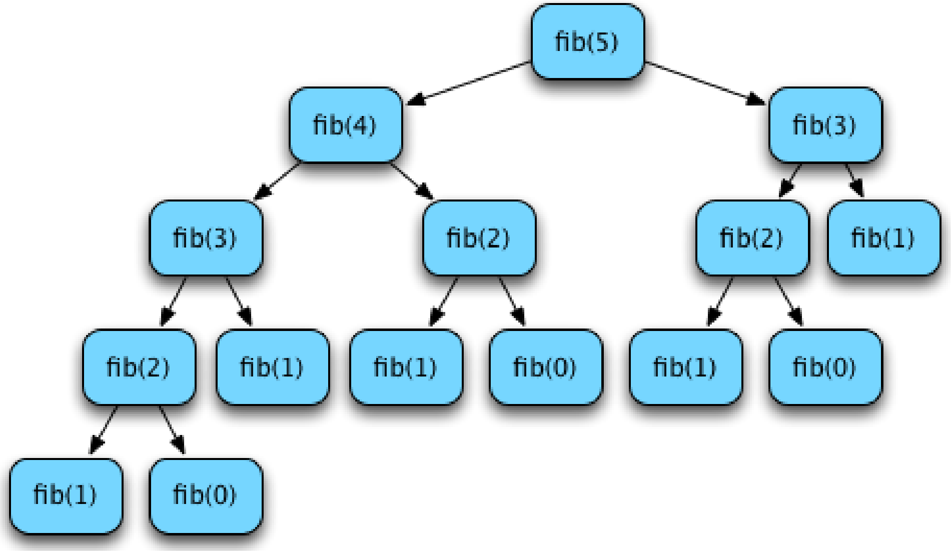
Dictionary operations

Memoization (1)

- **Memoization** – programming technique that improves **performance** of a function that is called **multiple times** with the same argument.
- **Memoization process:**
  - **Compute** a function with a **specific argument** only once & **record** the **result**.
  - **Return** recorded **result** when the function is called with the **same argument** again.
  - Result is stored in a **map**.
    - **Key** = **argument**, **value** = **result** of computing function with the argument.

Memoization (2)

- **Example:**
  - **Recursive Fibonacci** implementation requires invoking same calculations multiple times.
    - Computing fib(n) more than doubles the number of calls to compute fib(n-2).
    - **Exponential growth -> O(2^n)**.
  - **Memoized Fibonacci** implementation has **O(n) complexity**.
    - Fib is called with a new value of n -> answer is recorded in the map.
    - Fib(n) is called a subsequent time for some n -> memoized result is looked up and returned.



Fibonacci of 5 example

SUMMARY

- Sets.
- Hashing.
- Collision resolution.
- Maps.
- Memoization.