Python Cheat Sheet Made by Abdul Malik

💡 **Portfolio Website :** https://www.maliksquared.com/

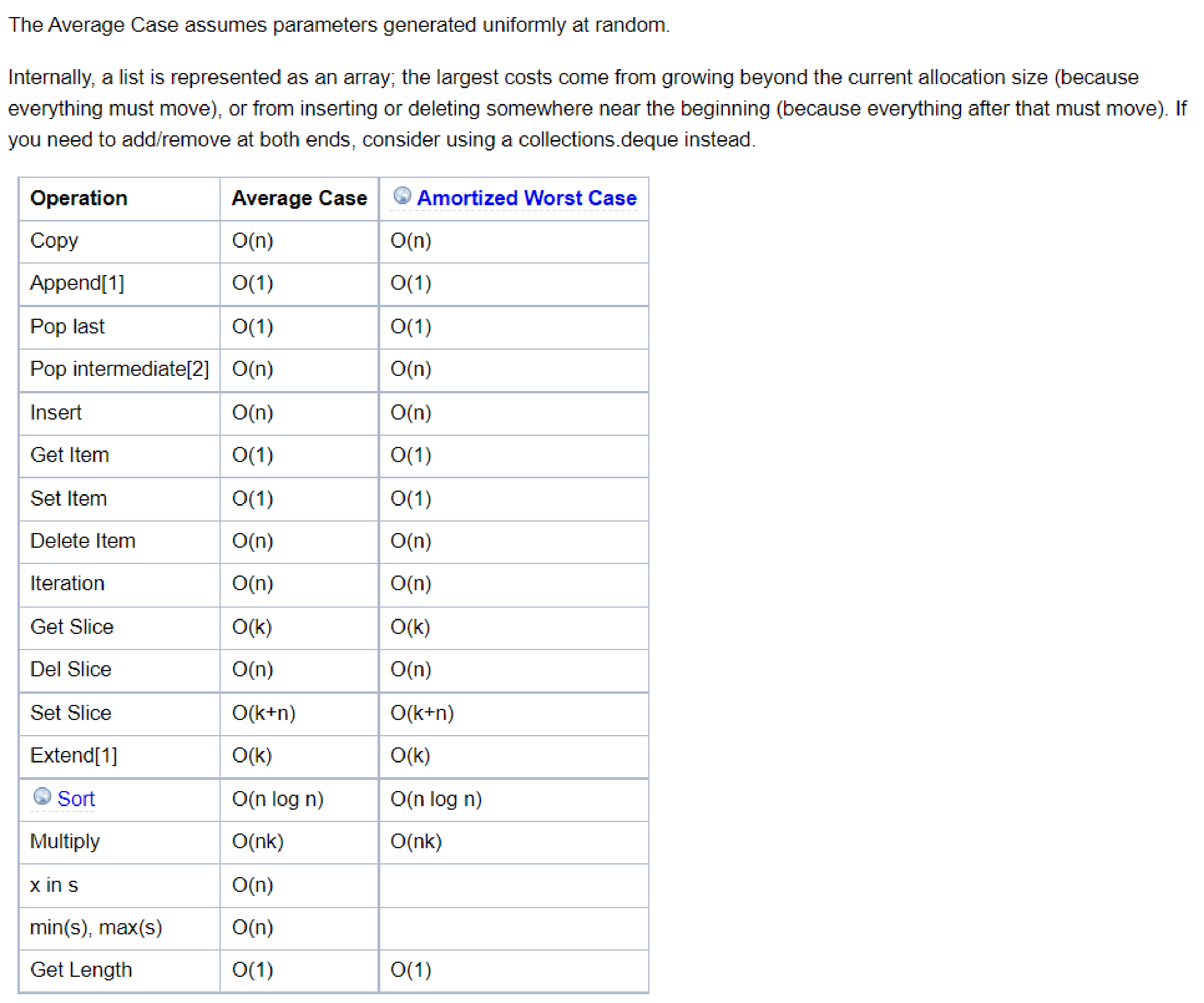
Data Structures

Important data structures for Leetcode

Lists

> Lists are used to store multiple items in a single variable

Operations Time Complexities



The Average Case assumes parameters generated uniformly at random.

Internally, a list is represented as an array; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a collections.deque instead.

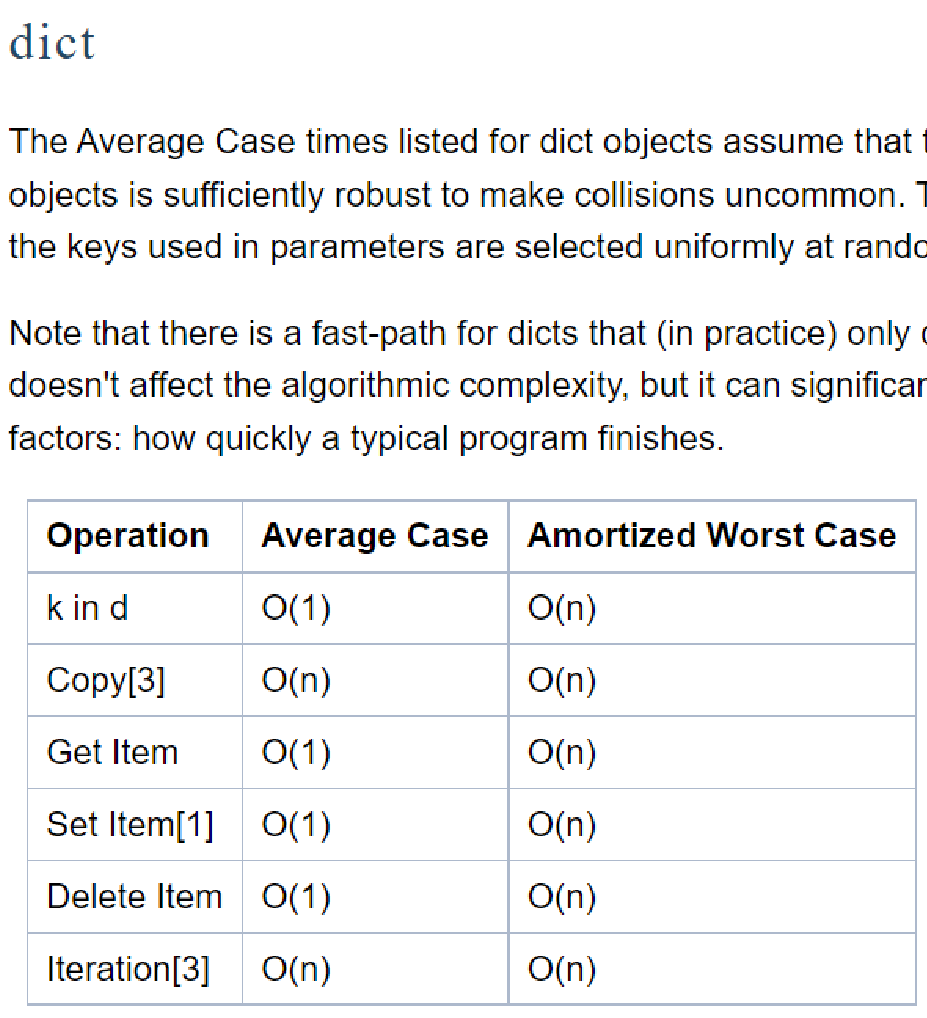
Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate[2]	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

```
nums = [1,2,3]

nums.index(1) # returns index
nums.append(1) # appends 1
nums.insert(0,10) # inserts 10 at 0th index
nums.remove(3) # removes all instances of 3
nums.copy(1) # returns copy of the list
nums.count(1) # returns no.of times '1' is present in the list
nums.extend(someOtherList) # ...
nums.pop() # pops last element [which element to pop can also be given as optional argument]
nums.reverse() # reverses original list (nums in this case)
nums.sort() # sorts list [does NOT return sorted list]
```

Dictionary

> Dictionaries are used to store data values in key:value pairs. *Info about **collections.Counter()** available below.*

Operations Time Complexities



dict

The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.

Note that there is a fast-path for dicts that (in practice) only deal with str keys; this doesn't affect the algorithmic complexity, but it can significantly affect the constant factors: how quickly a typical program finishes.

Operation	Average Case	Amortized Worst Case
k in d	O(1)	O(n)
Copy[3]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[3]	O(n)	O(n)

```
dict = {'a':1,'b':2,'c':3}

dict.keys() # returns list of keys of dictionary
dict.values() # returns list of values of dictionary
dict.get('a') # returns value for any corresponding key
dict.items() # returns [('a',1),('b',2),('c',3)]
dict.copy() # returns copy of the dictionary
# NOTE : items() Returns view object that will be updated with any future changes to dict
dict.pop(KEY) # pops key-value pair with that key
dict.popitem() # removes most recent pair added
dict.setDefault(KEY,DEFAULT_VALUE) # returns value of key, if key exists, else default value returned
# If the key exist, this parameter(DEFAULT_VALUE) has no effect.
# If the key does not exist, DEFAULT_VALUE becomes the key's value. 2nd argument's default is None.
dict.update({KEY:VALUE}) # inserts pair in dictionary if not present, if present, corresponding value is overriden (not key)
# defaultdict ensures that if any element is accessed that is not present in the dictionary
# it will be created and error will not be thrown (which happens in normal dictionary)
# Also, the new element created will be of argument type, for example in the below line
# an element of type 'list' will be made for a Key that does not exist
myDictionary = defaultdict(list)
```

Counter

> Python Counter is a container that will hold the count of each of the elements present in the container. The counter is a sub-class available inside the dictionary class. Specifically used for element frequencies

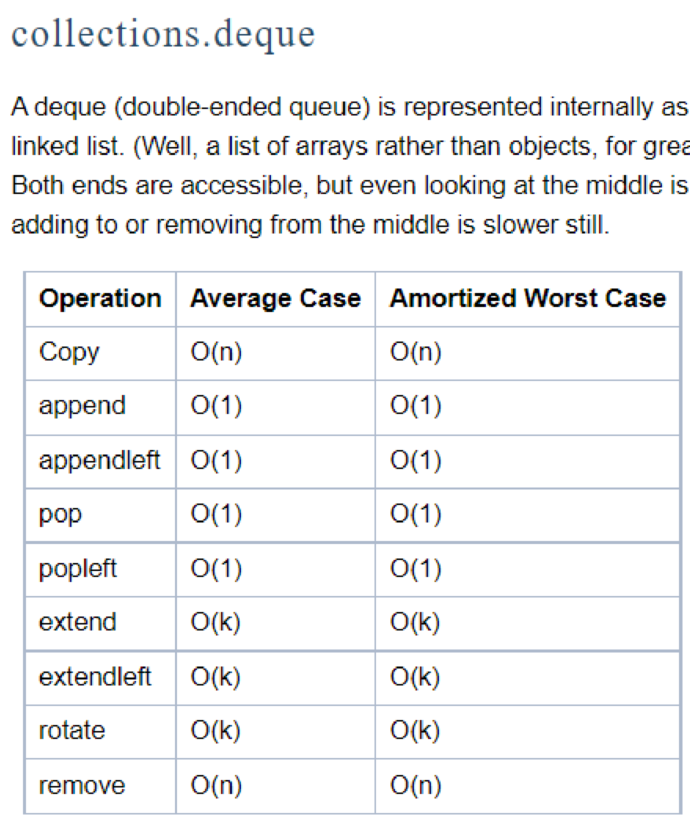
*Pretty similar to dictionary, infact I use **defaultdict(int)** most of the time*

```
from collections import Counter #(capital 'C')
# can also be used as 'collections.Counter()' in code

list1 = ['x','y','z','x','x','x','y', 'z']

# Initialization
Counter(list1) # => Counter({'x': 4, 'y': 2, 'z': 2})
Counter("Welcome to Guru99 Tutorials!") # => Counter({'o': 3, ' ': 3, 'u': 3, 'e': 2.....})

# Updating
counterObject = collections.Counter(list1)
counterObject.update("some string") # => Counter({'o': 3, 'u': 3, 'e': 2, 's': 2})
counterObject['s'] += 1 # Increase/Decrease frequency

# Accessing
frequency_of_s = counterObject['s']

# Deleting
del couterObject['s']
```

Deque

> A double-ended queue, or deque, has the feature of adding and removing elements from either end.

Operations Time Complexities



collections.deque

A deque (double-ended queue) is represented internally as a doubly linked list. (Well, a list of arrays rather than objects, for greater efficiency.) Both ends are accessible, but even looking at the middle is slow, and adding to or removing from the middle is slower still.

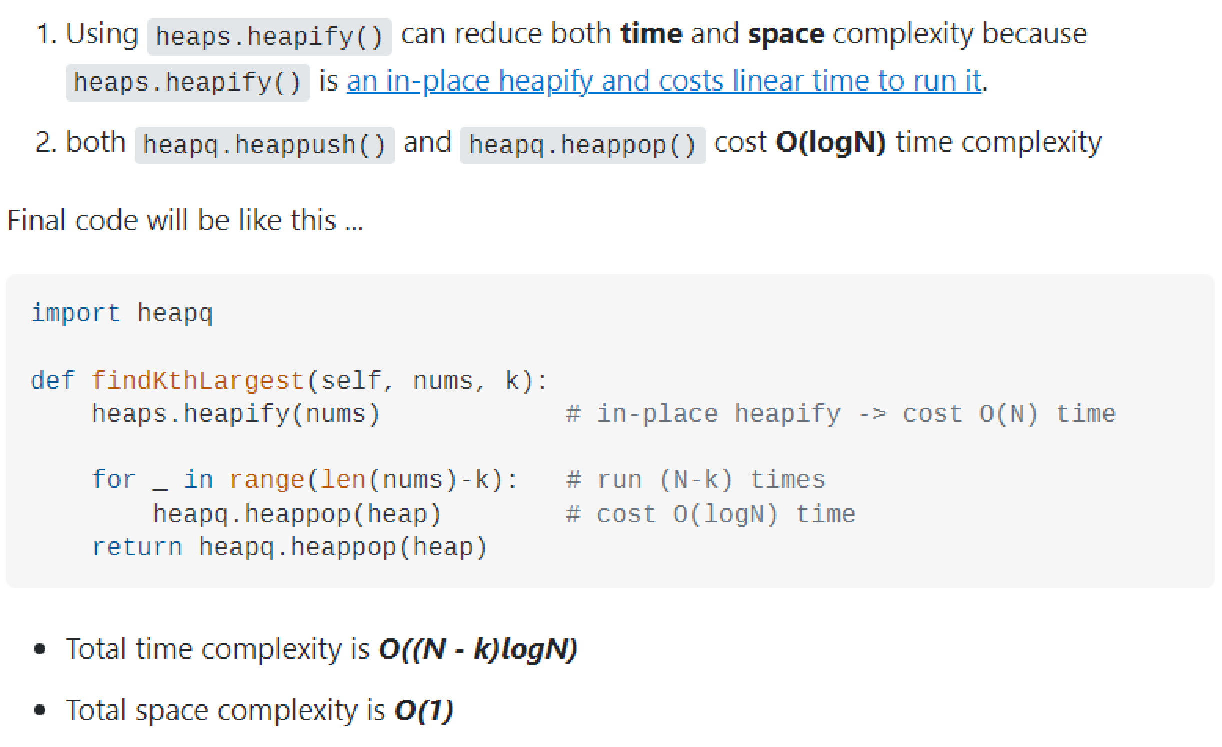
Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
append	O(1)	O(1)
appendleft	O(1)	O(1)
pop	O(1)	O(1)
popleft	O(1)	O(1)
extend	O(k)	O(k)
extendleft	O(k)	O(k)
rotate	O(k)	O(k)
remove	O(n)	O(n)

```
from collections import deque

queue = deque(['name','age','DOB'])

queue.append("append_from_right") # Append from right
queue.pop() # Pop from right

queue.appendleft("fromLeft") # Append from left
queue.popleft() # Pop from left

queue.index(element,begin_index,end_index) # Returns first index of element b/w the 2 indices.
queue.insert(index,element)
queue.remove() # removes first occurrance
queue.count() # obvious

queue.reverse() # reverses order of queue elements
```

Heapq

> As we know the Heap Data Structure is used to implement the Priority Queue ADT. In python we can directly access a Priority Queue implemented using a Heap by using the **Heapq** library/module.

Operations Time Complexities



1. Using `heaps.heapify()` can reduce both **time** and **space** complexity because `heaps.heapify()` is an in-place heapify and costs linear time to run it.
2. both `heapq.heappush()` and `heapq.heappop()` cost **O(logN)** time complexity

Final code will be like this ...

```
import heapq

def findKthLargest(self, nums, k):
    heaps.heapify(nums)            # in-place heapify -> cost O(N) time

    for _ in range(len(nums)-k):   # run (N-k) times
        heapq.heappop(heap)        # cost O(logN) time
    return heapq.heappop(heap)
```

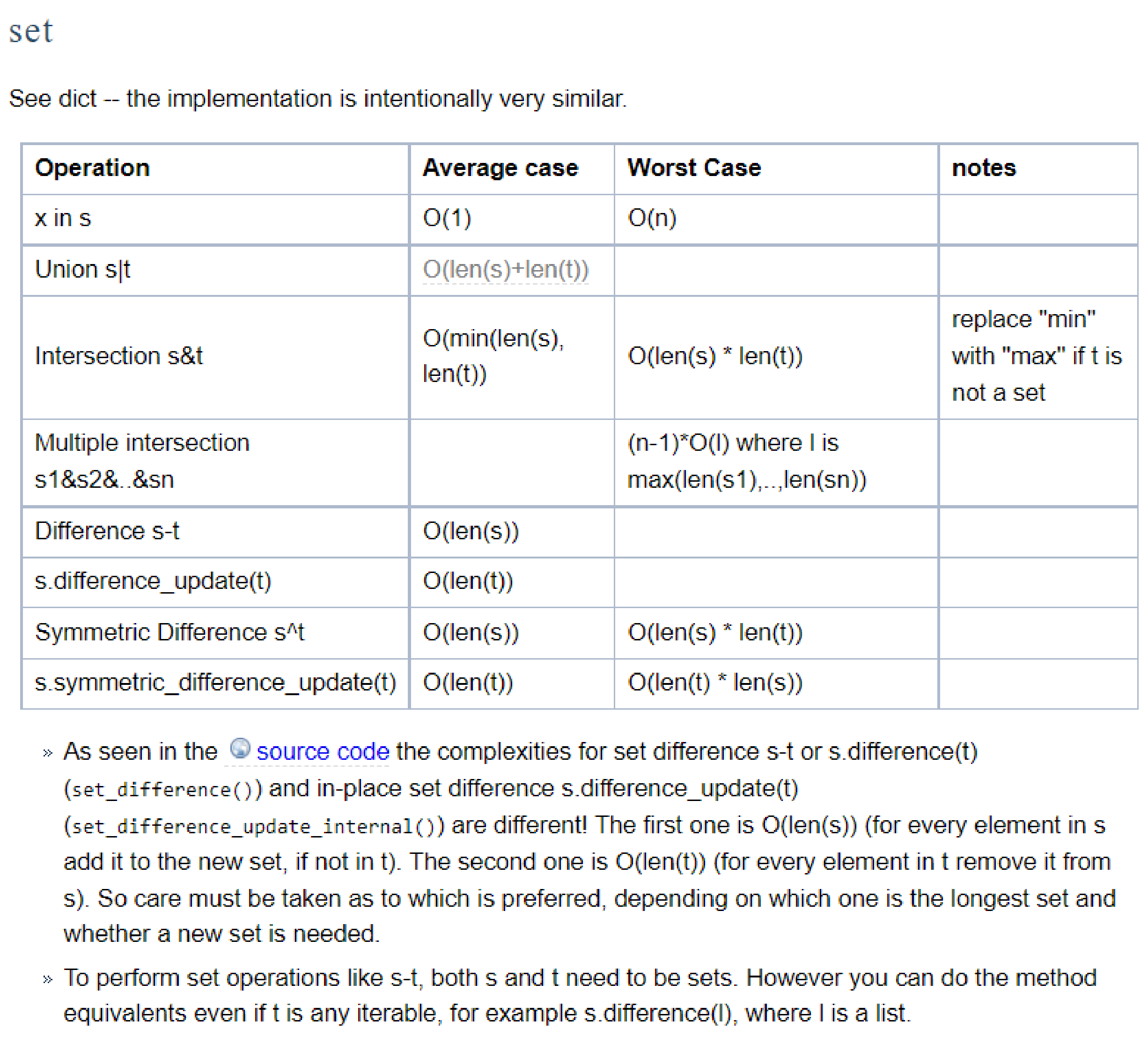
- Total time complexity is ***O((N - k)logN)***
- Total space complexity is ***O(1)***

```
import heapq # (minHeap by Default)

nums = [5, 7, 9, 1, 3]

heapq.heapify(nums) # converts list into heap. Can be converted back to list by list(nums).
heapq.heappush(nums,element) # Push an element into the heap
heapq.heappop(nums) # Pop an element from the heap
#heappush(heap, ele) :- This function is used to insert the element mentioned in its arguments into heap. The order is adjusted, so as heap structure is maintained.
#heappop(heap) :- This function is used to remove and return the smallest element from heap. The order is adjusted, so as heap structure is maintained.

# Rarely used methods
# Used to return the k largest elements from the iterable specified
# and satisfying the key (if is is mentioned).
heapq.nlargest(k, iterable, key = fun)
heapq.nsmallest(k, iterable, key = fun)
```

Sets

> A set is a collection which is unordered, immutable, unindexed, No Duplicates.

Operations Time Complexities



set

See dict -- the implementation is intentionally very similar.

Operation	Average case	Worst Case	notes
x in s	O(1)	O(n)	
Union s|t	O(len(s)+len(t))		
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))	replace "min" with "max" if t is not a set
Multiple intersection s1&s2&..&sn		(n-1)*O(l) where l is max(len(s1),..,len(sn))	
Difference s-t	O(len(s))		
s.difference_update(t)	O(len(t))		
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))	
s.symmetric_difference_update(t)	O(len(t))	O(len(t) * len(s))	

- As seen in the source code the complexities for set difference s-t or s.difference(t) (`set_difference()`) and in-place set difference s.difference_update(t) (`set_difference_update_internal()`) are different! The first one is O(len(s)) (for every element in s add it to the new set, if not in t). The second one is O(len(t)) (for every element in t remove it from s). So care must be taken as to which is preferred, depending on which one is the longest set and whether a new set is needed.
- To perform set operations like s-t, both s and t need to be sets. However you can do the method equivalents even if t is any iterable, for example s.difference(l), where l is a list.

```
set = {1,2,3}

set.add(item)
set.remove(item)
set.discard(item) | set.remove(item) # removes item | remove will throw error if item is not there, discard will not
set.pop() # removes random item (since unordered)

set.isdisjoint(anotherSet) # returns true if no common elements
set.issubset(anotherSet) # returns true if all elements from anotherSet is present in original set
set.issuperset(anotherSet) # returns true if all elements from original set is present in anotherSet

set.difference(anotherSet) # returns set containing items ONLY in first set
set.difference_update(anotherSet) # removes common elements from first set [no new set is created or returned]
set.intersection(anotherSet) # returns new set with common elements
set.intersection_update(anotherSet) # modifies first set keeping only common elements
set.symmetric_difference(anotherSet) # returns set containing all non-common elements of both sets
set.symmetric_difference_update(anotherSet) # same as symmetric_difference but changes are made on original set

set.union(anotherSet) # ...
set.update(anotherSet) # adds anotherSet without duplicate
```

Tuple

> A tuple is a collection which is ordered, unchangeable and can contain duplicate values

Operations Time Complexities

Similar to list

```
tuple = (1,2,3,1)

tuple.count(1) # returns occurence of an item
tuple.index(1) # returns index of 1 in array
```

Built-in or Library functions

```
** map(fun, iter) **
#fun : It is a function to which map passes each element of given iterable.
#iter : It is a iterable which is to be mapped.

** zip(list,list) **
for elem1,elem2 in zip(firstList,secondList):
  # will merge both lists and produce tuples with both elements
  # Tuples will stop at shortest list (in case of both lists having different len)

** any(list) ** [ OPPOSITE IS => ** all() ** ]
any(someList) # returns true if ANY element in list is true [any string, all numbers except 0 also count as true]

** enumerate(list|tuple) **
# [when you need to attach indexes to lists or tuples ]
enumerate(anyList) # ['a','b','c'] => [(0, 'a'), (1, 'b'), (2, 'c')]

** filter(function|list) **
filter(myFunction,list) # returns list with elements that returned true when passed in function

***************** import bisect ***********************

** bisect.bisect(list,number,begin,end) ** O(log(n))
# [ returns the index where the element should be inserted
#   such that sorting order is maintained ]
a = [1,2,4]
bisect.bisect(a,3,0,4) # [1,2,4] => 3 coz '3' should be inserted in 3rd index to maintain sorting order

# Other variants of this functions are => bisect.bisect_left() | bisect.bisect_right()
# they have same arguments. Suppose the element we want to insert is already present
# in the sorting list, the bisect_left() will return index left of the existing number
# and the bisect_right() or bisect() will return index right to the existing number

# ** bisect.insort(list,number,begin,end)       ** O(n) to insert
# ** bisect.insort_right(list,number,begin,end) **
# ** bisect.insort_left(list,number,begin,end)  **

The above 3 functions are exact same of bisect.bisect(), the only difference
is that they return the sorted list after inserting and not the index. The
left() right() logic is also same as above.
```

```
** ord(str) **
# returns ascii value of the character , Example ord("a") = 97
** chr(int) **
#return character of given ascii value , Example chr(97) = "a"
```