*Hash tables achieve constant time for search because everything is in order so it just traverses the table, O(1).
*Most common scrambling strategy is taking the ascii value of the item and dividing it by the size of the table.
*Collision is when an item is hashed to a slot that is already occupied in a hash table.
*Probing- when a hash value adds sequential numbers to the value until an open slot in the table is found
*Linear open addressing- method of collision resolution where the probe value is fixed, often at 1
*Hash table efficiency- O(1) if no collisions, O(n) if there is
*Dynamic programming- method for solving a complex problem by breaking it down into smaller problems and solving each of them, storing their solutions in a memory based database/data structure.
*Sequential search doesn't have to be sorted and to find an element you must traverse all the elements leading up to it. Binary search has to sort first, but it doesn't have to search all the elements.
*recursive algorithms are too slow when they have to constantly allocate big chunks of stack because of too many local variables (MergeSort)
*Memoization is when a function stores expensive function call results and returns them when the same input occurs again

Greedy Change-Making Algorithm:

```
def recursive_change(self, rem):
    c = Change(.71)
    c.recursive_change(c.convert)
    if len(self.coins) == 0:
        return []
        coin = self.coins.pop()
        num, new_rem = divmod(rem, coin)
        self.printer(num,coin)
    return self.recursive_change(new_rem) + [num
```

Recursion- when a function calls itself
Direct recursion- when a function invokes itself
Indirect recursion- when a function invokes another function, eventually invoking the first function again
Base case- condition that allows a function to stop recursing
System stack- place in memory for the things the heap doesn't cover
Stack frame- stores variables; When the function returns, the return value is left on top of the stack for the calling function to access.
Stack overflow- when too many function calls are made without returning

Recursive functions will terminate when the base case is satisfied.
Recursion can be a useful thing because it breaks complex problems down into much smaller problems that are easier to solve, then can use those solutions to help with the big problem.
Static storage- memory for your variables is allocated when the program starts
Automatic storage- usually stored on the stack; You do not have to reserve extra memory using them, but on the other hand, have also limited control over the lifetime of this memory
Dynamic storage- You now control the exact size and the lifetime of these memory locations. If you don't free it, you'll run into memory leaks, which may cause your application to crash, since it, at some point cannot allocate more memory
Logarithms with different bases only vary by a constant, so their time complexity is the same.
Exponentials with different bases grow exponentially but the higher the base, the faster the rate. Complexity wise they're the same.
Code for timing functions: import time, then time.time()
If an algorithm has an inferior time complexity, it can still be more useful than the better algorithm if
List/Set O(1)-
List/Set O(n)-
"size of the problem"- the number of items being operated on
Significant operation-the most nested operation
T(n) is measured by how long f(n) takes to finish
T(n) is O(f(n)) if it has the same complexity as f(n)
Node has value: head/tail pointer to connect to other nodes, public interface: addFirst(item), removeFirst(item), insertAfter(node), insertBefore(node), search(item), pop(node)
Infix: 25 * (x + y) – 7 + 8 * x
Postfix: 25 x y + * 7 – 8 x * +
Prefix: + (- (* (25, + (x, y)), 7), * (8, x))
Postfix evaluation algorithm converts expressions into postfix form
SingleNode has a value and tail pointers only, Singly Linked list
OrderedList inserts items into a sorted list, when added the items will be in order
Create a python class:
def cSort:
def bubbleSort(alist):

```
def bubbleSort(alist):
    for k in range(len(alist)-1,0,-1):
        for i in range(k):
            if alist[i]>alist[i+1]:
                temp=alist[i]
                alist[i]=alist[i+1]
                alist[i+1]=temp
    return(alist)
```

Stack- init, push, pop, top, size, IsEmpty
Queue- init, IsEmpty, enqueue, dequeue, size
Deque- init, size, addFront, removeFront, addRear, removeRear

```
def bubbleSort(alist):
    for k in range(len(alist)-1,0,-1):
        for i in range(k):
            if alist[i]>alist[i+1]:
                temp=alist[i]
                alist[i]=alist[i+1]
                alist[i+1]=temp
    return(alist)
def selectionSort(alist):
    for j in range(len(alist)-1,0,-1):
        aMax=0
        for k in range(1,j+1):
            if alist[k]>alist[aMax]:
                aMax=k
        temp=alist[j]
        alist[j]=alist[aMax]
        alist[aMax]=temp
    return(alist)
```

InsertionSort:It always maintains a sorted sublist in the lower positions of the list. Each new item is then "inserted" back into the previous sublist such that the sorted sublist is one item larger.
MergeSort:Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted. Then it goes back and merges the small sublist together and keeps doing it until you have one large sorted list
QuickSort:quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

Subscripting- selects an item of a dict, set, list
Slicing- selects a segment of a list, string, or set
Pass messages to items in a tuple:
tupl = ([], 5, True)
tupl[0].append(12)
mutable- dict, set, bytearray, lists
immutable- int, float, str, bytes, tuple
abstraction- data hiding
encapsulation- data hiding
interface-
data hiding-
client- user of code, writing more code
mutable- item can be changed
immutable- item cannot be changed
keyed- assigned values to a key
redefining the __lt__() function changes the sorted order from ascending to descending
items in a set or dict have no order
a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.

```
class TreeNode:
    def __init__(self, key, value = None):
        self.key = key
        self.value = value
        self.parent = self.leftChild = self.rightChild = None
```

```
fin = open("file.txt","r")
line = fin.readline()
while line != " ":
    line=fin.readline()

def bubbleSort(alist):
  for k in range(len(alist)-1,0,-1):
    for i in range(k):
      if alist[i]>alist[i+1]:
        temp=alist[i]
        alist[i]=alist[i+1]
        alist[i+1]=temp
  return(alist)
def selectionSort(alist):
  for j in range(len(alist)-1,0,-1):
    aMax=0
    for k in range(1,j+1):
      if alist[k]>alist[aMax]:
        aMax=k
    temp=alist[j]
    alist[j]=alist[aMax]
    alist[aMax]=temp
  return(alist)
```

Subscripting- selects an item of a dict, set, list
Slicing- selects a segment of a list, string, or set
Pass messages to items in a tuple:
tupl = ([], 5, True)
tupl[0].append(12)
mutable- dict, set, bytearray, lists
immutable- int, float, str, bytes, tuple
abstraction- data hiding
encapsulation- data hiding
interface-
data hiding-
client- user of code, writing more code
mutable- item can be changed
immutable- item cannot be changed
keyed- assigned values to a key
redefining the __lt__() function changes the sorted order from ascending to descending
items in a set or dict have no order
a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.

```
class TreeNode:
    def __init__(self, key, value = None):
        self.key = key
        self.value = value
        self.parent = self.leftChild = self.rightChild = None
```

Binary Search Trees:
Parent node has no more than two children
Balanced binary tree- all levels are full
"getting"- traverse the tree to the top from the node you're searching for, return the value
"putting"- traverse the tree down to the correct position for the value, places it there, creates new connections and a new level for the nodes it moves down

Adjacency Matrix representation:

	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	0	0

Adjacency List Representation: