1.

```
import random
import time

class HashTable:
    def __init__(self):
        self.size = 99
        self.slots = [None] * 100
        self.data = [None] * 100

    def quickLoad(self, values, load):
      load = (load / 100) + 1 #Determines load in percent
      self.size = int(len(values) * load) #Size
      '''Creates list'''
      self.slots = [None] * self.size
      self.data = [None] * self.size
      '''Creates keys, if wanted.'''
      self.keys = []

      '''Loop creates the hashtable for every value in the list it was give'''
      for i in range(len(values)):
        key = 0
        '''This checks to see if the itme is a string, if so, it generates a key
that is the hash values added for each character.'''
        if type(values[i]) is str:
          for char in values[i]:
            key += ord(char)
        else:
          '''Otherwise, the key will just equal the integer value of the item in
the list.'''
          key = int(values[i])
        self.keys.append(key)
        self.put(key,values[i])


    def put(self,key,data):
      hashvalue = self.hashfunction(key,len(self.slots))

      if self.slots[hashvalue] == None:
        self.slots[hashvalue] = key
        self.data[hashvalue] = data
      else:
        if self.slots[hashvalue] == key:
          self.data[hashvalue] = data  #replacea
        else:
```

```
        nextslot = self.rehash(hashvalue,len(self.slots))
        while self.slots[nextslot] != None and \
                        self.slots[nextslot] != key:
          nextslot = self.rehash(nextslot,len(self.slots))

        if self.slots[nextslot] == None:
          self.slots[nextslot]=key
          self.data[nextslot]=data
        else:
          self.data[nextslot] = data #replace

    def hashfunction(self,key,size):
         return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size

    def get(self,key):
      startslot = self.hashfunction(key,len(self.slots))

      data = None
      stop = False
      found = False
      position = startslot
      while self.slots[position] != None and  \
                           not found and not stop:
         if self.slots[position] == key:
           found = True
           data = self.data[position]
         else:
           position=self.rehash(position,len(self.slots))
           if position == startslot:
               stop = True
      return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)

def main():
  A=HashTable()
  B=HashTable()
  TOTLIST = random.sample(range(1000000), 200000)
```

```
  INS = TOTLIST[:10000]
  NOTINS = TOTLIST[10000:]

  '''This loop executes for list items up to 100000 in 10000 increments.'''
  for LEN in range(10000, 100001, 10000):
    '''This creates the items in the myINS and myNOTINS, and creates two hash
tables for it every time around.'''
    A.quickLoad(INS[:LEN], 50)
    B.quickLoad(INS[:LEN], 95)
    myINS = INS[:LEN]
    myNOTINS = NOTINS[:LEN]

    '''Each of these for loops is to get the time for how long it takes to find
each individual item in the list in it's range. This has to be done as you cannot
pass an entire list to search.'''
    start = time.time()
    for i in range(len(myINS)):
      print(A[myINS[i]])
    end = time.time()

    aTimeINS =+ end - start

    start = time.time()
    for i in range(len(myNOTINS)):
      print(A[myNOTINS[i]])
    end = time.time()

    aTimeNOTINS =+ end - start

    start = time.time()
    for i in range(len(myINS)):
      print(B[myINS[i]])
    end = time.time()

    bTimeINS =+ end - start

    start = time.time()
    for i in range(len(myNOTINS)):
      print(B[myNOTINS[i]])
    end = time.time()

    bTimeNOTINS =+ end - start

    start = time.time()
    for i in range(len(myINS)):
```

```
      print(linear_search(myINS, myINS[i]))
    end = time.time()

    linSearchINS =+ end - start

    start = time.time()
    for i in range(len(myINS)):
      print(linear_search(myINS, myNOTINS[i]))
    end = time.time()

    linSearchNOTINS =+ end - start

    start = time.time()
    for i in range(len(myINS)):
      print(binary_search(myINS, myINS[i]))
    end = time.time()

    binSearchINS =+ end - start

    start = time.time()
    for i in range(len(myINS)):
      print(binary_search(myINS, myNOTINS[i]))
    end = time.time()

    binSearchNOTINS =+ end - start


  '''This is to finish finding the average (dividing by ten).'''
  aTimeINS = (aTimeINS / 10)
  aTimeNOTINS = (aTimeNOTINS / 10)
  bTimeINS = (bTimeINS / 10)
  bTimeNOTINS = (bTimeNOTINS / 10)
  linSearchINS = (linSearchINS / 10) / 60
  linSearchNOTINS = (linSearchNOTINS / 10)
  binSearchINS = (binSearchINS / 10)
  binSearchNOTINS = (binSearchNOTINS / 10)

  print("The time it took for INS to find in a Hash with 50 percent load was " +
str(aTimeINS))
  print("The time it took for NOTINS to find in a Hash with 50 percent load was "
+ str(aTimeNOTINS))
  print("The time it took for INS to find in a Hash with 95 percent load was " +
str(bTimeINS))
  print("The time it took for NOTINS to find in a Hash with 95 percent load was "
+ str(bTimeNOTINS))
```

```
  print("The time it took for INS to find in a linear search was " +
str(linSearchINS))
  print("The time it took for NOTINS to find in  a linear search was " +
str(linSearchNOTINS))
  print("The time it took for INS to find in a binary search was: " +
str(binSearchINS))
  print("The time it took for NOTINS to find in a binary search was: " +
str(binSearchNOTINS))


'''This two functions are the ones provided. Thank you for making our lives a bit
easier.'''
def linear_search(mylist, find):
  for x in mylist:
    if x == find:
      return True
  return False

def binary_search(mylist, find):
  while len(mylist) > 0:
    mid = (len(mylist))//2
    if mylist[mid] == find:
      return True
    elif mylist[mid] < find:
      mylist = mylist[:mid]
    else:
      mylist = mylist[mid + 1:]
  return False

if __name__ == "__main__":
  main()
```

I used the linear collision method for rehash. The re-hash method was included in the hash class distributed to us. It takes the hash of the item and just adds 1 to it.

2.

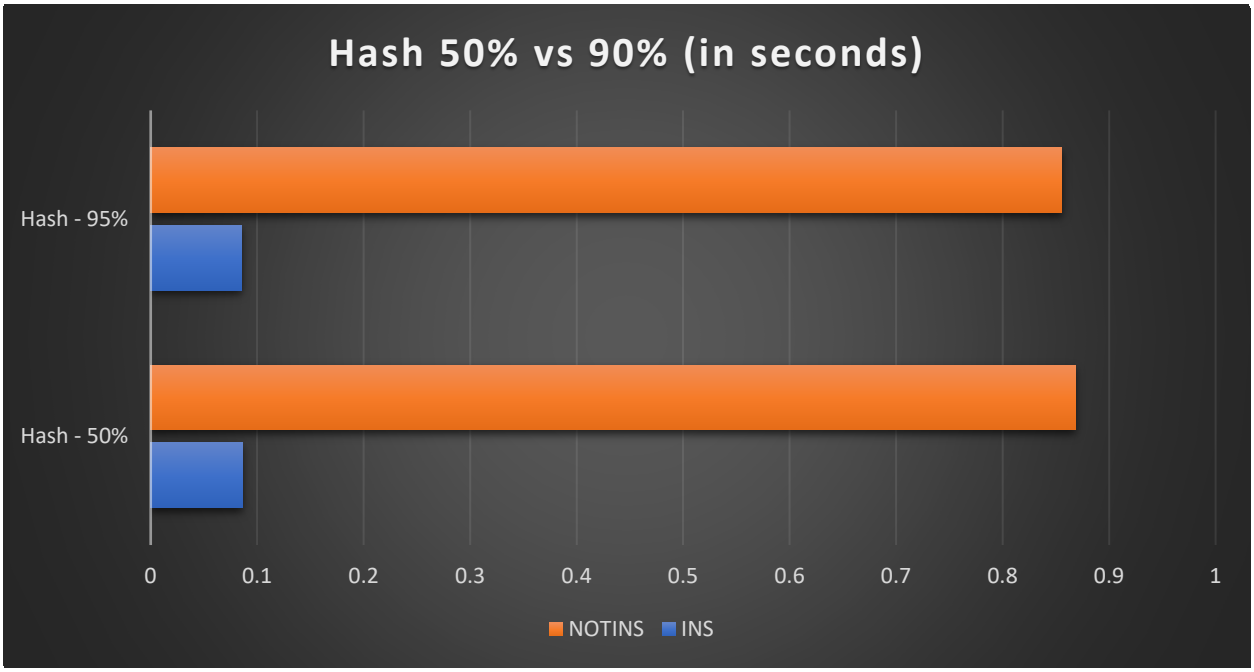
Test results for hashing with 95 and 50 percent loads:

The time it took for INS to find in a Hash with 50 percent load was 0.08607831001281738

The time it took for NOTINS to find in a Hash with 50 percent load was 0.8680923223495484

The time it took for INS to find in a Hash with 95 percent load was 0.0852776288986206

The time it took for NOTINS to find in a Hash with 95 percent load was 0.8555972576141357



3.

Output:

The time it took for INS to find in a Hash with 50 percent load was 0.08607831001281738

The time it took for NOTINS to find in a Hash with 50 percent load was 0.8680923223495484

The time it took for INS to find in a Hash with 95 percent load was 0.0852776288986206

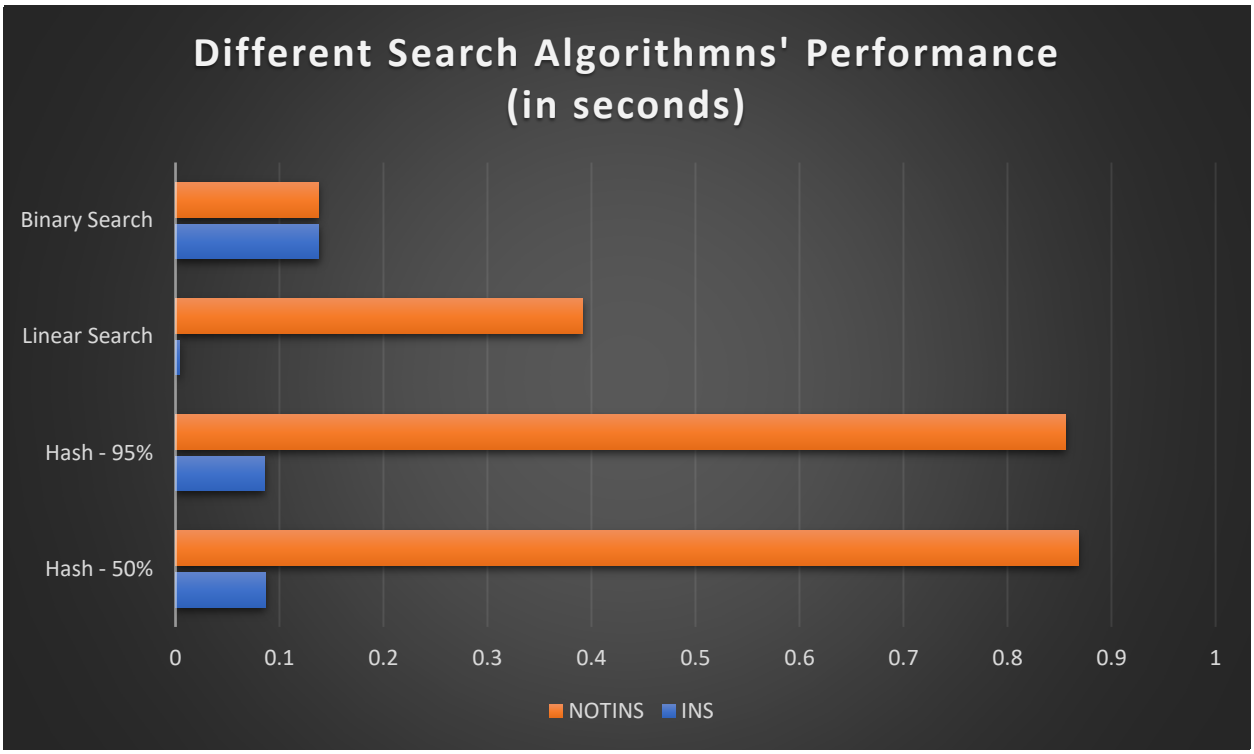
The time it took for NOTINS to find in a Hash with 95 percent load was 0.8555972576141357

The time it took for INS to find in a linear search was 0.004097059965133667

The time it took for NOTINS to find in a linear search was 0.3912558078765869

The time it took for INS to find in a binary search was: 0.13792550563812256

The time it took for NOTINS to find in a binary search was: 0.13736822605133056



4.

Question 1: Will hashing perform the quickest out of all the search methods?

Hypothesis: Yes, but only for the IN searches.

Question 2: Will linear searching perform better than binary search?

Hypothesis: Yes, in every instance the binary search should do better.

I believe that the linear search did good in overall time as it probably did not have to look very far for some of the items, but as you can see it has the second-best time for the NOTINS. The binary search was the overall best in this test, and I believe that is because it can exponentially reduce it's search time, even when the item is not in the list. The hashes logically performed better than the binary search, however, I am unsure how they did worse than linear for INS. I understand that the NOTINS would take longer, but I would've thought to see linear search take far longer than it did, and I would have thought to see the NOTINS for the hash take the same time as the linear search. I feel this most likely has something to do with taking the averages.