1. Consider the following sequential search (linear search) code:

Textbook's Listing 5.1

```
def sequentialSearch(alist, item):
    """ Sequential search of unorder list """
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1

    return found
```

Faster sequential search code

```
def linearSearch(aList, target):
    """Returns the index of target in aList
       or -1 if target is not in aList"""
    for position in range(len(aList)):
        if target == aList[position]:
            return position
    return -1
```

a) What is the *basic operation* of a search?

b) For the following `aList` value, which `target` value causes `linearSearch` to loop the fewest ("best case") number of times?

	0	1	2	3	4	5	6	7	8	9	10
aList:	10	15	28	42	60	69	75	88	90	93	97

c) For the above `aList` value, which `target` value causes `linearSearch` to loop the most ("worst case") number of times?

d) For a *successful search* (i.e., `target` value in `aList`), what is the "average" number of loops?

Textbook's Listing 5.2

```
def orderedSequentialSearch(alist, item):
    """ Sequential search of order list """
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1

    return found
```

Faster sequential search code

```
def linearSearchOfSortedList(target, aList):
    """Returns the index position of target in
    sorted aList or -1 if target is not in aList"""
    breakOut = False
    for position in range(len(aList)):
        if target <= aList[position]:
            breakOut = True
            break

    if not breakOut:
        return -1
    elif target == aList[position]:
        return position
    else:
        return -1
```

e) The above version of linear search assumes that `aList` is sorted in ascending order. When would this version perform better than the original `linearSearch` at the top of the page?

2. Consider the following binary search code:

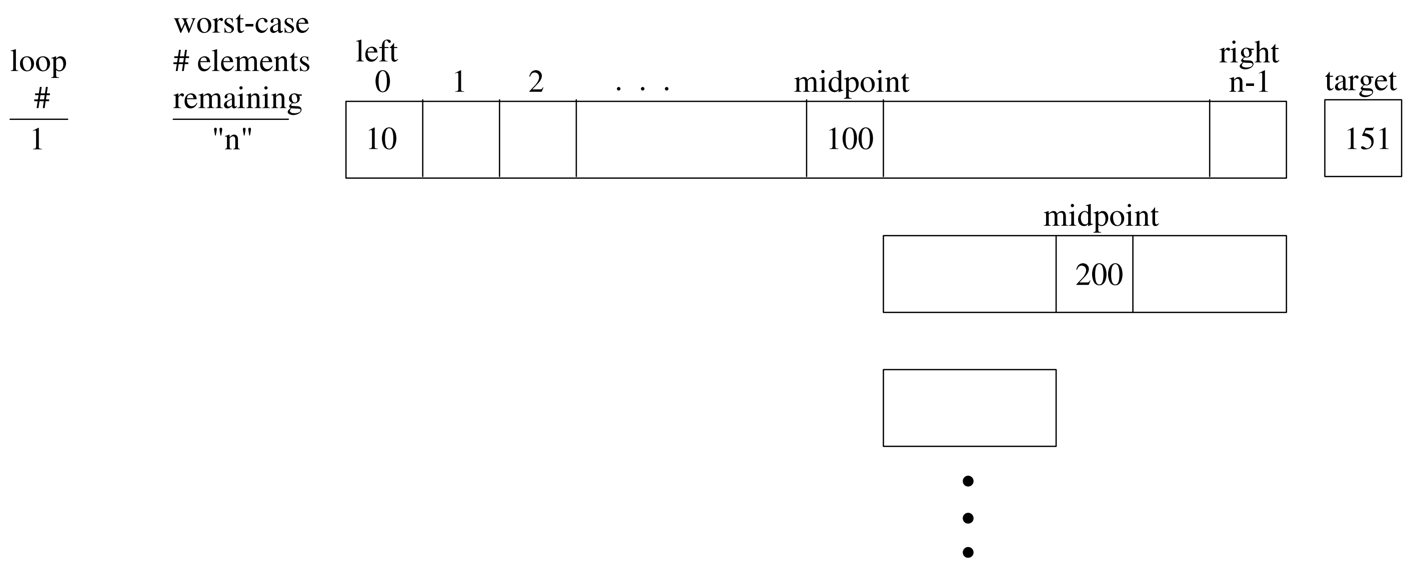
Textbook's Listing 5.3

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found
```

Faster binary search code

```
def binarySearch(target, lyst):
    """Returns the position of the target
       item if found, or -1 otherwise."""
    left = 0
    right = len(lyst) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if target == lyst[midpoint]:
            return midpoint
        elif target < lyst[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```

a) "Trace" binary search to determine the worst-case basic total number of comparisons?



b) What is the worst-case big-oh for binary search?

c) What is the best-case big-oh for binary search?

d) What is the average-case (expected) big-oh for binary search?

e) If the list size is 1,000,000, then what is the maximun number of comparisons of list items on a *successful search*?

f) If the list size is 1,000,000, then how many comparisons would you expect on an *unsuccessful search*?

3. Hashing Motivation and Terminology:

a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.

If n is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?

average case $O($ $\quad\quad\quad)$

worst case $O($ $\quad\quad\quad)$

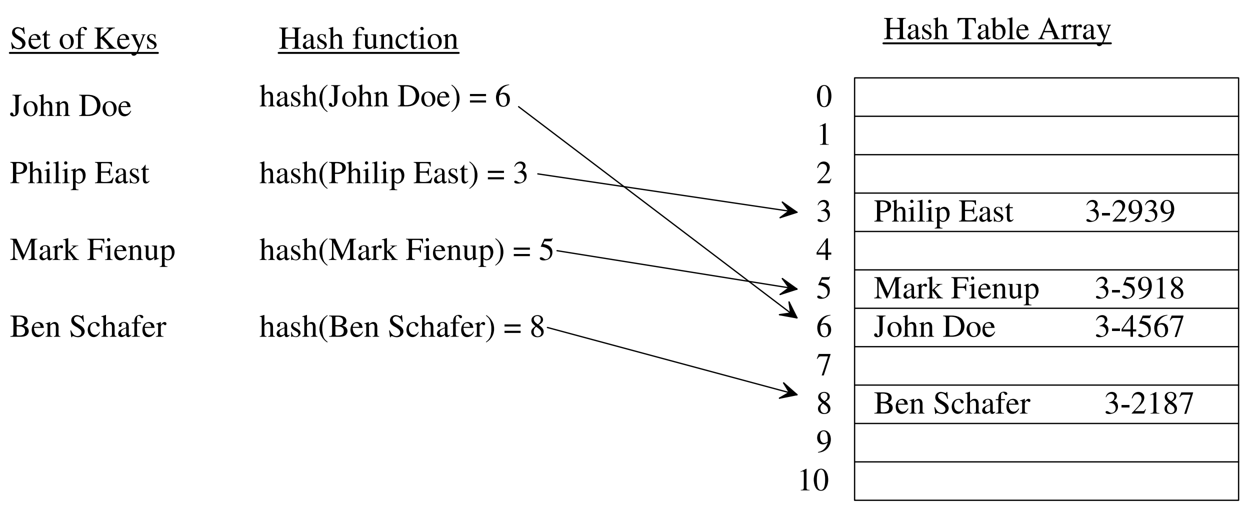
b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

If n is the number of items being searched, what is the average and worst case big-oh notation for a search?

average case $O($ $\quad\quad\quad)$

worst case $O($ $\quad\quad\quad)$

Hashing tries to achieve average constant time (i.e., $O(1)$) searching by using the target's value to calculate where in the array/Python list (called the *hash table*) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target's *home address*) is the job of the *hash function*. A *perfect hash function* would take your set of target values and map each to a unique array index.



a) If n is the number of items being searched and we had a perfect hash function, what is the average and worst case big-oh notation for a search?

average case $O($ $\quad\quad\quad)$

worst case $O($ $\quad\quad\quad)$

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same hash-table index, called a *collision*.

Collisions are handled by two approaches:

- *open-address* with some *rehashing* strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.

- *chaining, closed-address*, or *external chaining*: all target values hashed to the same home address are stored in a data structure (called a *bucket*) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)

5. Consider the following examples using *open-address* approach with a simple rehashing strategy of *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 8
Paul Gray (3-5917)	hash(Paul Gray) = 3
Sarah Diesburg (3-3-7395)	hash(Sarah Diesburg) = 3

Hash Table Array

0		
1		
2		
3	Philip East	3-2939
4		
5	Mark Fienup	3-5918
6	John Doe	3-4567
7		
8	Ben Schafer	3-2187
9		
10		

a) Assuming open-address with linear probing where would Paul Gray and then Sarah Diesburg be placed?

Common rehashing strategies include the following.

Rehash Strategy	**Description**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)
quadratic probing	Check the square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))//2) % (hash table size), where the hash table size is a power of 2
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size

b) Assume quadratic probing, insert "Paul Gray" and "Sarah Diesburg" into the hash table.

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 0
Paul Gray (3-5917)	hash(Paul Gray) = 3
Sarah Diesburg (3-7395)	hash(Sarah Diesburg) = 3

Hash Table Array

0	Ben Schafer	3-2187
1		
2		
3	Philip East	3-2939
4		
5	Mark Fienup	3-5918
6	John Doe	3-4567
7		

c) Assume double hashing, insert "Paul Gray" and "Sarah Diesburg" into the hash table.

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 0
Paul Gray (3-5917)	hash(Paul Gray) = 3 rehash_offset(Paul Gray) = 1
Sarah Diesburg (3-7395)	hash(Sarah Diesburg) = 3 rehash_offset(Sarah Diesburg) = 3

Hash Table Array

0	Ben Schafer	3-2187
1		
2		
3	Philip East	3-2939
4		
5	Mark Fienup	3-5918
6	John Doe	3-4567
7		

d) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Sarah Diesburg if the table was full? For the above example, (home address + (rehash attempt #) * offset) % (hash table size) would be: (3 + (rehash attempt #) * 3) % 8

Rehash Attempt #	0	1	2	3	4	5	6	7	8	9	10
Address											

e) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- *primary clustering* - keys mapped to a home address follow the same rehash pattern
- *secondary clustering* - rehash patterns from initially different home addresses merge together

Rehash Strategy	**Description**	**Suffers from:** primary clustering	**Suffers from:** secondary clustering
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)		
quadratic probing	Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2		
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size		

6. Let λ be the *load factor* (# item/hash table size). The average probes with **linear probing** for insertion or unsuccessful search is: $\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\lambda)^2}\right)\right)$. The average for successful search is: $\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\lambda)}\right)\right)$.

a) Why is an unsuccessful search worse than a successful search?

The average probes with **quadratic probing** for insertion or unsuccessful search is: $\left(\frac{1}{1-\lambda}\right) - \lambda - \log_e(1-\lambda)$

The average probes with quadratic probing for successful search is: $1 - \left(\frac{\lambda}{2}\right) - \log_e(1-\lambda)$

Consider the following table containing the average number probes for various load factors:

Probing Type	**Search outcome**	**Load Factor, λ**				
		0.25	**0.5**	**0.67**	**0.8**	**0.99**
Linear Probing	unsuccessful	1.39	2.50	5.09	13.00	5000.50
	successful	1.17	1.50	2.02	3.00	50.50
Quadratic Probing	unsuccessful	1.37	2.19	3.47	5.81	103.62
	successful	1.16	1.44	1.77	2.21	5.11

b) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

7. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 8
Paul Gray	hash(Paul Gray) = 3
Sarah Diesburg	hash(Sarah Diesburg) = 4

Hash Table Array

0		
1		
2		
3	Philip East	3-2939
4	Paul Gray	3-5917
5	Mark Fienup	3-5918
6	John Doe	3-4567
7	Sarah Diesburg	3-7395
8	Ben Schafer	3-2187
9		
10		

a) If "Mark Fienup" is deleted, how will we find Sarah Diesburg?

b) How might we fix this problem?