1 Lists

To construct a list, one can use the `list` constructor, so `l = list()` returns an empty list. The constructor also takes any *iterable* object in Python and constructs a list from it. For example `list(range(5))` returns a new list equal to `[0,1,2,3,4]` and `list("cow")` returns a new list equal to `['c','o','w']`. One can use the square bracket notation to create lists too, so `[3,1,4,1,5,9]` returns an appropriate list of length 6.

Operations

Lists, like strings, are sequences of objects, so they support the sequence operations:

- indexing,
- slicing, and
- length.

These operations are *not side-effecting*—they won't affect the old list—however, there are many differences between lists and strings:

- Lists are *mutable*, which means that we can change the contents of the list several of its methods. If `l` is a list, then the following operations are all popular methods for manipulating `l`:

  index assignment `l[i] = obj` means replace the object at index `i` of `l` with `obj`.

  appending `l.append(obj)` means append `obj` to `l` so that the length of `l` increases by one.

  inserting `l.insert(i,obj)` means insert `obj` at index `i` of `l`; the length of the list increases by one.

  popping `l.pop(i)` means delete the the object at index `i`; `l.pop()` means delete the last object.

  deleting `del l[i]` means delete the object at index `i` of `l`; this decreases the length of the list by one.

  removing `l.remove(obj)` means remove the first item in `l` that equals `obj`.

- Sort lists using the `sort()` method.
- Lists are *heterogenous*, which means they can simultaneously store objects of different type.
- Lists are really *adjustable arrays*, which we will examine in detail later.
- Lists support *list comprehensions*, which allow you to make new lists from other iterables. For example, to generate the first five non-negative multiples of 5, one could write:

```
[5*i for i in range(10)]
```

Let `l = list(range(10))`. What does `l` equal after the following operations?

```
l.append(11)
del l[0]
l.remove(1)
```

Let `l = list('sub pop')`. What does `l` equal after the following operations?

```
l.insert(3,'*')
l[len(l)-2] = 'u'
l.append('!')
l.append(l.pop())
```

2 Searching

A fundamental operation in computer science in *search*. Suppose we have the following list of strings:

```
l = ["The Strokes", "Bon Iver", "Arcade Fire", "The Black Keys",
     "Pixies", "The White Stripes", "Neutral Milk Hotel",
     "The National", "Yo La Tengo"]
```

If we call the sort method `l.sort()` then the list `l` of strings becomes:

```
l = ['Arcade Fire', 'Bon Iver', 'Neutral Milk Hotel', 'Pixies',
     'The Black Keys', 'The National', 'The Strokes',
     'The White Stripes', 'Yo La Tengo']
```

Notice that `sort` is *side-effecting*—it changes the current list. The `sort` method returns `None` so you should not use it where you expect a return value. If you want a new sorted list, then call `sorted(l)` which makes a copy of `l`, sorts it, and then returns it.

Suppose we want to be able to find a string in the list that begins with a certain prefix.
Call this function `find_startswith(last,searchstr)` and consider it's natural definition below:

```
1  def find_startswith(lst,searchstr):
2      '''linear search through lst to find the first string starting with searchstr'''
3      for s in lst:
4          if s.startswith(searchstr):
5              return s
6      return None
```

Question 1. *In the worst case, if* `lst` *has* n *elements, how many elements will* `find_startswith` *examine?*

Can we do better?

```
1  def find_startswith(lst, searchstr):
2      low = 0
3      high = len(lst)-1
4      while (low <= high):
5          mid = (high + low) // 2
6          if lst[mid].startswith(searchstr):
7              return lst[mid]
8          elif lst[mid] < searchstr:
9              low = mid+1
10         else:
11             high = mid-1
12     return None
```

Question 2. *In the worst case, if* `lst` *has* n *elements, how many elements will* `find_startswith` *examine?*