Lab 4

Exercise 1: Implement `selectionSort`, `insertionSort`, `bubbleSort`, and `mergeSort`. Try hard to do it without looking at the lecture notes, and just by implementing the basic ideas of each method:

- `selectionSort`: Find the smallest item, swap it with the item at the beginning, then recursively sort the rest.
- `insertionSort`: Iteratively make sure L[0:i] is sorted, starting with i=1, and going up t i=`len`(L).
- `bubbleSort`: In each iteration, move from left to right in the list, swapping when necessary. Terminate when no swaps are left.
- `mergeSort`: Recursively sort the left and right halves of the list, then merge the results.

Now, experiment with the different sorting methods. In particular, try executing the following:

```
>>> n = 100
>>> L = range(n)
>>> L.reverse()
>>> X = bubbleSort(L)
```

Note that after the sort has completed, you'll be given back the prompt `>>>`. As long as the prompt isn't given back yet, the sort algorithm is still executing. Try the same code with `n = 10000`. Notice how long it takes for different sorting algorithms besides just `bubbleSort`. Then try it with `n = 100000`. Also try it without the `L.reverse()` line (the `reverse()` function for the `list` data type reverse the elements in the `list`).

Exercise 2: Implement a function `hasElementSum`(n, L) where n is an `int` and L is a `list` of `int`s. The function should return `False` if no two distinct elements in L sum to n, and otherwise it should return a list of size two, where the elements of the returned `list` are two elements in L which sum to n. There can be multiple valid return values. Your algorithm should be able to handle lists of size up to one million. Use the fact that `mergeSort` can quickly sort `list`s of size up to one million.

For example, `hasElementSum`(5,[1,2,3,4]) could either return [1,4], or [4,1], or [2,3], or [3,2]. `hasElementSum`(8,[1,2,3,4]) should return `False` (4+4 is 8, but you shouldn't use the same element twice). `hasElementSum`(4,[2]) should return `False` since two distinct elements don't sum to 4 (we don't even have two elements), but `hasElementSum`(4,[2,2]) should return [2,2].

Slow example solution:

```
def hasElementSum(n, L):
for i in xrange(len(L)):
for j in xrange(i+1, len(L)):
if L[i]+L[j] == n:
return [L[i], L[j]]
return False
```

If you run the above code on lists of size 10000 or so, you'll already start to notice the code running slowly. The next implementation however can run on lists that have millions of elements.

Fast example solution:

```
def binarySearch(n, a, b, L):
    # check if n is in the sorted list L, somewhere between index a and b
    if a>b:
        return False
    mid = (a+b)/2
    if L[mid] == n:
        return True
    elif L[mid] > n:
        return binarySearch(n, a, b-1, L)
    else:
        return binarySearch(n, mid+1, b, L)

def hasElementSum(n, L):
    L = mergeSort(L)
    for i in xrange(len(L)):
        v = L[i]
        if v+v==n:
            if (i>0 and L[i-1]==v) or (i+1<len(L) and L[i+1]==v):
                return [v, v]
            else:
                continue
        elif binarySearch(n-v, 0, len(L)-1, L):
            return [v, n-v]
    return False
```

There is an even faster way to solve this problem using a technique known as *hashing*, but it is beyond the scope of this course.