1 Administrative Topics

- The research symposium is beginning today, and I am chairing a session from 1-2:30 (though I am not sure the last talk is supposed to be that long). And the computer science talks are from 2:45 to 4pm. So, I will be in my office after 4.

2 Recursion

We reminded ourselves that recursion is a technique in which functions call themselves. A recursive function has a base case and a recursive case. Recursive solutions to problems break the problem into one small task plus the rest of the problem (which can be solved by applying the same algorithm to a subproblem).

2.1 Reversing a String

We want to use recursion to reverse the characters in a string.

The technique we will use is to break the string into two parts – the last character and the “rest of the characters”. If we recursively reverse the rest of the characters and move the last character to the beginning, then will will reverse the string.
So, if our string is “abc”, then we need to reverse ”bc” and move ’a’ to the end (i.e. reverse(”bc”) + ”a” is our answer).

The base cases are for strings of length 0 and 1. In those cases, the reverse of a string is just the string itself.

The code is here:

```python
# Return a copy of s in reverse order
# (s must be a string)
def reverse(s):
    if len(s) <= 1:
        # base case (list and reverse are same)
        return s
    else:
        # recursive case
        return s[-1] + reverse(s[:-1])
    #return reverse(s[1:]) + s[0]
```

Note that the last line provides an alternative algorithm – we break the string into the first character and the “rest of the characters”. In this case, we move the first character to the end.

We test this function with the following code:

```python
def testReverse():
    data = [‘abc’, ’’, ’a’, ’abcd’]
    for datum in data:
        print datum + ” reversed is ” + reverse(datum)
```

And it works. The code is very deceptive because it looks like it doesn’t do anything. The key is that each call to reverse accomplishes a small part of the overall task (it moves the first character to the end or the last character to the beginning), but it is called lots of times. Together, all the calls accomplish the entire task.

### 2.2 Binary Search

The binary search example demonstrates that recursion can be used to make searching more efficient.

Suppose we have a sorted list of numbers (e.g. [-100, 1, 2, 3, 100]) and we want to see if 5 is in the list.
We could perform a linear search – start at the beginning and check each element, breaking out of the loop when we find 5 is in the list or are convinced it is not in the list (i.e. when we encounter an element that is bigger than 5). And this is a fantastic strategy for short lists. But for long lists, this can take too long. Why? Because if the list is of length N, then, on average, we will need to examine N/2 elements. And in the worst case, we will need to examine N elements.

What if we use an algorithm that takes better advantage of the sorted nature of the list. The binary search algorithm does just that – it repeatedly splits the list in half, searching smaller and smaller sublists for an element.

Let’s run through the same example. We are looking for 5 in [-100, 1, 2, 3, 100]. The first thing we do is check the middle element (which is 2). Is 2 == 5? No, 5 > 2, so 5 must be in the second half of the list. Now, let’s search for 5 in the second half of the list (i.e. [3, 100]). Let’s say the middle element of this list is 100. Is 5 == 100? No, 5 < 100, so 5 must be in the first half of this shortened list. Now, let’s search of 5 in the first half of this list (i.e. [3]). This list has only one element (3). Is 3 == 5? No. So, 5 must not be in this list.

Notice that we are repeating the same algorithm on a smaller and smaller list. Recursion is a natural approach to take to implement the binary search.

There are two base cases: the list is empty and the list is of length 1.

The remaining code checks the middle element, and then searches the appropriate half of the list (if necessary).

Here it is:

```python
# return True if goal is an element in lst
# return False otherwise
# NOTE: The lst must be SORTED!!!
def binarySearch(lst, goal):
    print lst
    # base cases
    if len(lst) == 0:
        return False
    elif len(lst) == 1:
        return goal == lst[0]
    # recursive cases
    mididx = len(lst)/2
    print "mididx is " + str(mididx) + " and value is " + str(lst[mididx])
```

3
```python
if lst[mididx] == goal:
    return True
elif lst[mididx] < goal:
    # search the right half of the list
    return binarySearch(lst[mididx+1:], goal)
else:
    return binarySearch(lst[:mididx], goal)
```

We test it with this code:

```python
def testBinarySearch():
data = [([], 2), ([1], 1), ([1], 0), (range(9), 0), (range(9), 1), (range(9), 2), (range(9), 3), (range(9), 4), (range(9), 5), (range(9), 6), (range(9), 7), (range(9), 8), (range(9), 9), (range(9), 10)]
for datum in data:
    print str(datum[1]) + " in " + str(datum[0]) +": " + \s
    print str(binarySearch(datum[0], datum[1]))
```