CS 120: Intro to Computer Programming II

In-Class Activity - 05 Recursion, part 2 - Day 2

Remember how I told you to keep your solutions from the previous activity? Go get them, you'll be using them!

Activity 1 - Turn in this one

In the previous meeting, we used a loop to simulate recursion; each iteration of the loop sliced one value off of the array. It ran in $O(n^2)$ time.

Here's the recursive version. It will fail if you make `N` even moderately large; can your group figure out why?

```
def sumlist_slice_1(vals):
    if len(vals) == 0:
        return 0
    return vals[0] + sumlist_slice_1(vals[1:])
```

Solution: This code will hit a stack overflow if `N` is only a few thousand.

Activity 2 - Turn in this one

There's a good way to avoid the stack overflow from the previous activity: divide things up by half, instead of slicing only one at a time. Write a new, recursive version of the function, named `sumlist_sliceHalf()`, which breaks the input data in half, and recurses into both halves.

Once `sliceHalf` appears to be working correctly, start testing it to see how fast it is. Assuming that you're doing slicing of the array like I've asked, you should find that it is **slower** than $O(n)$ - but it should be **much better** than the $O(n^2)$ version we looked at in the previous meeting.

Solution: TODO

Activity 3 - Turn in this one

A trick, to avoid the cost of slicing, is to pass the **entire array** - but to use parameters which give the start and the end of the range you want to consider.

In a recursive function, it's often handy to have a "wrapper" which doesn't have these parameters - which your users can call - and then a **different** function inside, which is actually recursive.

In this Activity, rewrite the `sliceHalf` function above to avoid slicing entirely - but otherwise, keep its logic entirely intact. I've provided the public "wrapper" function - you need to provide the private, recursive function that it calls.

```
def sumlist_pretend_sliceHalf(vals):
    return _sumlist_pretend_sliceHalf(vals, 0,len(vals))

def _sumlist_pretend_sliceHalf(vals, start,end):
    YOUR CODE HERE
```

Notice:

- The private version has the same name, except with a leading underscore. This isn't required, but it's a common style choice.
- The start parameter is inclusive, but the end is exclusive.

Solution:

```
def sumlist_pretend_sliceHalf(vals):
    return _sumlist_pretend_sliceHalf(vals, 0,len(vals))

def _sumlist_pretend_sliceHalf(vals, start,end):
    if end-start == 0:
        return 0
    if end == start+1:
        return vals[end]

    mid = (start+end) // 2       # this is equivalent to: start + (end-start)//2
    return _sumlist_pretend_sliceHalf(vals, start,mid) +
           _sumlist_pretend_sliceHalf(vals,       mid,end)
```

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Now, let's compare the performance of the two versions of `sliceHalf`. We've already seen that the performance of the version that actually did slicing (from Activity 2) was $O(n \lg n)$. What is the performance of the one that just **pretends** to slice?

Solution: It is $O(n)$.

Activity 5 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Remember binary search? It was another of the examples that we used for recursion. Write a version of binary search which is **recursive,** but which uses this "pretend slicing" trick to work more efficiently.

What is its performance?

Solution:

```
def binary_search(data, val):
    return _binary_search(data, val, 0,len(data))

def _binary_search(data, val, start,end):
    if start==end:
        return -1      # not found

    mid = (start+end) // 2
    if data[mid] == val:
        return mid

    if val < data[mid]:
        return _binary_search(data, val, start,mid)
    else:
        return _binary_search(data, val,       mid+1,end)
```

Activity 6 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Finally, let's take the recursive version of binary search, from the previous step, and turn it into a looping version. Don't use recursion in this one - instead, just use a `while` loop.

Confirm that it runs in $O(\lg n)$ time, just like the recursive version.

Solution:

```
def binary_search(data, vals):
    start = 0
    end   = len(data)

    while start < end:
        mid = (start+end) // 2
        if data[mid] == val:
            return mid

        if val < data[mid]:
            end = mid
        else:
            start = mid+1

    assert start == end
    return -1             # fail
```