1 Administrative Topics

- We look at Proj 9 images.

2 Recursion

A powerful design technique in computer science is recursion – a technique in which a function calls itself. Recursion is useful for large problems that can be broken down into a solution to one small piece of the problem plus the solution to the rest of the problem.

2.1 Factorial

To understand recursion more detail, let’s examine it in the context of mathematical problems.

A classic example is that of the factorial.

What is the solution to 6! ? It is the solution to 6 * 5!. What is the solution to 5!? It is the solution to 5 * 4! and this pattern continues until we stop at 0!.

Let’s try coding up a function that will solve the factorial problem.

% Return n!
def factorial(n):

Using the description I gave above, what is the central idea?

\[ n! = n \times (n - 1)! \]

So, let’s put that in code

```python
% Return n!
def factorial(n):
    ret = n * factorial(n-1)
```

What will happen if we try to run this code as is? It will never end, because we don’t have any stopping condition. What is a good stopping condition for factorial? It is when \( n=0 \). \( 0! \) is 1, so let’s use that fact to fix our code.

```python
% Return n!
def factorial(n):
    if n == 0:
        ret = 1
    else:
        ret = n * factorial(n-1)
    return ret
```

Now, this will work, as long as \( n \geq 0 \) when we first call factorial.

What should we do if the input is not an integer? I think we should just let it die (i.e. we don’t write code to handle it). But what if it is a negative integer? We could add code that make it return None if it is \( \leq 0 \).
Now, to the memory model.

We begin with a main function, which wants to print the result of factorial (3):

The function begins execution and its symbol table is set up with \( n \) as 3. Since \( n \) is not zero, it enters the recursive case (passing in 2 as the value for \( n \)).
The function begins execution and its symbol table is set up with \( n \) as 2. Likewise, it enters the recursive case, passing in 1 as the value for \( n \):
The function begins execution and its symbol table is set up with \( n \) as 1. Likewise, it enters the recursive case, passing in 0 as the value for \( n \):
The function begins execution and its symbol table is set up with n as 0. This time, it enters the base case – no more function calls will be made. Woo Hoo! We have reached the “bottom” of the recursion.
Now begin the process of returning from each of the functions. First, the base case function returns 1 to its caller (marked in red).
Its caller can now compute its value of `ret`:

```
main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial</td>
<td></td>
</tr>
<tr>
<td>print factorial(3)</td>
<td></td>
</tr>
</tbody>
</table>

factorial

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>3 &lt;int&gt;</td>
</tr>
</tbody>
</table>

ret = 3 * factorial(2)

factorial

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2 &lt;int&gt;</td>
</tr>
</tbody>
</table>

ret = 2 * factorial(1)

factorial

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1 &lt;int&gt;</td>
</tr>
</tbody>
</table>

ret = 1 * factorial(0) 1
```
And return its value:
And the process continues going “up” the stack of function calls.
And another return value is sent “up”:
And another return value can be computed:

And finally, the value is returned to the main function:
Notice that the stack of function tables grew, as each function passed a smaller version of the problem to another copy of the function. It stopped when we reached the smallest problem (0!), and then the stack shrank as the solution to the smaller problems propagated back up to the original caller.

2.2 General rules for recursion

- The function must have a “base case” (there is no recursive call).
- The function must make a recursive call on a problem that is slightly smaller. E.g. in the case of the factorial, if the factorial function is called with input $N$, then the recursive call is given input $N-1$.

2.3 Other Examples

2.3.1 Summing the Elements of a List

We can use recursion to operate on lists as well. For example, let’s write a function that sums the entries of a list. The strategy we take is to add the first element of the list to the sum of the remaining elements. We turn the problem into a smaller and smaller problem, by finding the sum of shorter and shorter lists. Our base case is a list of length 0.

The code is:

```python
# return the sum of all the elements in the list.
def sum(lst):
    if len(lst) == 0:
        # base case
        return 0
    else:
        # recursive case
        return lst[0] + sum(lst[1:])
```

13
2.3.2 Summing the Odd Elements of a List

We change the above example so that it checks to see if the first element is odd. If it is, it add it to the result of the recursive call. If it isn’t it just returns the result of the recursive call, i.e.

```python
# return the sum of all the odd elements (I don’t
# means odd-indexed elements, I mean odd values)
def sumOdd(lst):
    if len(lst)==0:
        return 0
    elif lst[0] % 2 == 1:  # odd
        return lst[0] + sumOdd(lst[1:]))
    else:  # even
        return sumOdd(lst[1:])
```

2.3.3 Summing Every Other Element of a List

We change yet again to sum just the odd-indexed elements. Our strategy is to lop off two elements from the list when passing it to the recursive case. This introduces the need for a second base case – one for a list of length 0 (if the original list has an even number of entries) and one for a list of length 1 (if the original list has an odd number of entries).

The code is:

```python
# Return the sum of every other element in the
# list
def sumAlternates( lst ):
    if len(lst) == 0:
        return 0
    elif len(lst) == 1:
        return lst[0]
    else:
        return lst[0] + sumAlternates(lst[2:])
```