CS103 Unit 8

Recursion

Mark Redekopp
Recursion

• Defining an object, mathematical function, or computer function in terms of *itself*

GNU
• Makers of gedit, g++ compiler, etc.
• GNU = GNU is Not Unix
  GNU is Not Unix
    GNU is Not Unix
      GNU is Not Unix
        ... is Not Unix is not Unix is Not Unix
Recursion

- Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) and a base/terminating case
- Usually takes the place of a loop
- Input to the problem must be categorized as a:
  - Base case: Solution known beforehand or easily computable (no recursion needed)
  - Recursive case: Solution can be described using solutions to smaller problems of the same type
    - Keeping putting in terms of something smaller until we reach the base case

- Factorial: n! = n * (n-1) * (n-2) * ... * 2 * 1
  - n! = n * (n-1)!
  - Base case: n = 1
  - Recursive case: n > 1 => n*(n-1)!
Recursive Functions

- Recall the system stack essentially provides separate areas of memory for each ‘instance’ of a function.
- Thus each local variable and actual parameter of a function has its own value within that particular function instance’s memory space.

C Code:

```c
int fact(int n)
{
    // base case
    if(n == 1)
        return 1;

    // recursive case
    else {
        // calculate (n-1)!
        int small_ans = fact(n-1);
        // now ans = (n-1)!
        // so calculate n!
        return n * small_ans;
    }
}
```
Recursive Call Timeline

- Value/version of n is implicitly “saved” and “restored” as we move from one instance of the ‘fact’ function to the next.

```c
int fact(int n)
{
    if(n == 1)
        return 1;
    else {
        int small_ans = fact(n-1);
        return n * small_ans;
    }
}
```
Head vs. Tail Recursion

• Head Recursion: Recursive call is made before the real work is performed in the function body
• Tail Recursion: Some work is performed and then the recursive call is made

Tail Recursion

```cpp
void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        cout << "Go" << endl;
        doit(n-1);
    }
}
```

Head Recursion

```cpp
void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        doit(n-1);
        cout << "Go" << endl;
    }
}
```
Head vs. Tail Recursion

- **Head Recursion**: Recursive call is made before the real work is performed in the function body
- **Tail Recursion**: Some work is performed and then the recursive call is made

```cpp
Void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        cout << "Go" << endl;
        doit(n-1);
    }
}
```

```
doit(3)
  Go
  doit(2)
    Go
    doit(1)
      Stop
      return
  return
  return
  return
  Go
  return
  Go
  Stop
```

```
Void doit(int n)
{
    if(n == 1) cout << "Stop";
    else {
        doit(n-1);
        cout << "Go" << endl;
    }
}
```

```
doit(3)
  Go
  doit(2)
    Go
    doit(1)
      Stop
      return
  return
  return
  Go
  return
  Go
```

Recursive Functions

- Recall the system stack essentially provides separate areas of memory for each ‘instance’ of a function
- Thus each local variable and actual parameter of a function has its own value within that particular function instance’s memory space

C Code:

```c
int main()
{
    int data[4] = {8, 6, 7, 9};
    int sum1 = isum_it(data, 4);
    int sum2 = rsum_it(data, 4);
}

int isum_it(int data[], int len)
{
    int sum = data[0];
    for(int i=1; i < len; i++){
        sum += data[i];
    }
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum = rsum_it(data, len-1);
        return sum + data[len-1];
}
```
Recursive Call Timeline

int main()
{
    int data[4] = {8, 6, 7, 9};
    int sum2 = rsum_it(data, 4);
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum = rsum_it(data, len-1);
        return sum + data[len-1];
}

Each instance of rsum_it has its own len argument and sum variable

Every instance of a function has its own copy of local variables
The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function.

```c
int main()
{
    int data[4] = {8, 6, 7, 9};
    int sum2 = rsum_it(data, 4);
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum =
            rsum_it(data, len-1);
        return sum + data[len-1];
}
```

---

*System Stack & Recursion*

**System Memory (RAM)**

<table>
<thead>
<tr>
<th>Code for all functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data for rsum_it (data=800, len=1, sum=??) and return link</td>
</tr>
<tr>
<td>Data for rsum_it (data=800, len=2, sum=8) and return link</td>
</tr>
<tr>
<td>Data for rsum_it (data=800, len=3, sum=14) and return link</td>
</tr>
<tr>
<td>Data for rsum_it (data=800, len=4, sum=21) and return link</td>
</tr>
<tr>
<td>Data for main (data=800,sum2=??) and return link</td>
</tr>
</tbody>
</table>

---

```
data[4]: 8 6 7 9
800
```
Exercise

- Exercises
  - Count-down
  - Count-up
Recursion Double Check

• When you write a recursive routine:
  – Check that you have appropriate base cases
    • Need to check for these first before recursive cases
  – Check that each recursive call makes progress toward the base case
    • Otherwise you'll get an infinite loop and stack overflow
  – Check that you use a 'return' statement at each level to return appropriate values back to each recursive call
    • You have to return back up through every level of recursion, so make sure you are returning something (the appropriate thing)
Loops & Recursion

• Is it better to use recursion or iteration?
  – ANY problem that can be solved using recursion can also be solved with iteration and other appropriate data structures

• Why use recursion?
  – Usually clean & elegant. Easier to read.
  – Sometimes generates much simpler code than iteration would
  – Sometimes iteration will be almost impossible
  – The power of recursion often comes when each function instance makes multiple recursive calls

• How do you choose?
  – Iteration is usually faster and uses less memory
  – However, if iteration produces a very complex solution, consider recursion
Recursive Binary Search

- Assume remaining items = [start, end)
  - start is inclusive index of start item in remaining list
  - End is exclusive index of start item in remaining list

- binSearch(target, List[], start, end)
  - Perform base check (empty list)
    - Return NOT FOUND (-1)
  - Pick mid item
  - Based on comparison of k with List[mid]
    - EQ => Found => return mid
    - LT => return answer to BinSearch[start, mid)
    - GT => return answer to BinSearch[mid+1, end)
Sorting

• If we have an unordered list, sequential search becomes our only choice
• If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
• Many sorting algorithms of differing complexity (i.e. faster or slower)
• Bubble Sort (simple though not terribly efficient)
  – On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)
Exercise

• Exercises
  – Text-based fractal
Flood Fill

- Imagine you are given an image with outlines of shapes (boxes and circles) and you had to write a program to shade (make black) the inside of one of the shapes. How would you do it?
- Flood fill is a recursive approach
- Given a pixel
  - Base case: If it is black already, stop!
  - Recursive case: Call floodfill on each neighbor pixel
  - Hidden base case: If pixel out of bounds, stop!