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: \( n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \)
  – \( n! = n \times (n-1)! \)
  – Base case: \( n = 1 \)
  – Recursive case: \( n > 1 \Rightarrow n \times (n-1)! \)
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 n_less_one = fact(n-1);
        // now ans = (n-1)!
        // so calculate n!
        return = n * n_less_one;
    }
}
```
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 n_less_one = fact(n-1);
        return n * n_less_one ;
    }
}
```
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);
    }
}
```

```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.

---

### Tail Recursion

```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
      - Go
    - Stop
  - Return
    - Go
    - Stop
    - Return
      - Go
      - Stop
\n
### Head Recursion

```cpp
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
      - Go
    - Stop
    - Go
  - Stop
  - Go
  - Stop
```

---
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
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)
{
    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

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.

```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];
}
```

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

```c
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];
}
```

```c
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

- 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 =
            sum_them(data, len-1);
    return sum + data[len-1];
}
```

System Memory (RAM)

- Data for `rsum_it` (data=800, len=1, sum=??) and return link
- Data for `rsum_it` (data=800, len=2, sum=8) and return link
- Data for `rsum_it` (data=800, len=3, sum=14) and return link
- Data for `rsum_it` (data=800, len=4, sum=21) and return link
- Data for `main` (data=800, size=4, sum1=??, sum2=??) and return link

System stack area

```
800
8 6 7 9
```

data[4]: 0 1 2 3
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

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

• Exercises
  – Text-based fractal
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)`

List: [2, 3, 4, 6, 9, 11, 13, 15, 19]
index: 0 1 2 3 4 5 6 7 8
start: i
end: i
k = 11

List: [2, 3, 4, 6, 9, 11, 13, 15, 19]
index: 0 1 2 3 4 5 6 7 8
start: i
end: i
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!
Analyze These!

- What does this function print?

```c
void f2(int n) {
    if (n > 0) {
        cout << *;
        f2(n-1);
    }
}

void f1(int m, int n){
    if( m > 0 ){
        f2(n);
        cout << endl;
        f1(m-1, n);
    }
}

int main() {
    f1(4, 3);
}
```
Analyze These!

• What does this function print?

```cpp
void rfunc(int n, int t) {
    if (n == 0) {
        cout << t << " ";
        return;
    }
    rfunc(n-1, 3*t);
    rfunc(n-1, 3*t+2);
    rfunc(n-1, 3*t+1);
}
int main() {
    rfunc(2, 0);
}
```

• What does this function return for g(3122013)

```cpp
int g(int n) {
    if (n % 2 == 0)
        return n/10;
    return g(g(n/10));
}
```
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)
Bubble Sort Algorithm

\[ n \leftarrow \text{length(List)}; \]
\[ \text{for} \ (i=n-2; \ i \geq 1; \ i--) \]
\[ \text{for} \ (j=1; \ j \leq i; \ j++) \]
\[ \text{if} \ (\ \text{List}[j] > \text{List}[j+1]) \ \text{then} \]
\[ \text{swap List}[j] \text{ and List}[j+1] \]

Pass 1

\[
\begin{array}{cccccc}
7 & 3 & 8 & 6 & 5 & 1 \\
j & i \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{swap} \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{no swap} \\
3 & 7 & 6 & 8 & 5 & 1 \\
\text{swap} \\
3 & 7 & 6 & 5 & 8 & 1 \\
\text{swap} \\
3 & 7 & 6 & 5 & 1 & 8 \\
\text{swap} \\
\end{array}
\]

Pass 2

\[
\begin{array}{cccccc}
3 & 7 & 6 & 5 & 1 & 8 \\
j & i \\
3 & 6 & 7 & 5 & 1 & 8 \\
\text{no swap} \\
3 & 6 & 7 & 5 & 1 & 8 \\
\text{swap} \\
3 & 6 & 5 & 7 & 1 & 8 \\
\text{swap} \\
3 & 6 & 5 & 1 & 7 & 8 \\
\text{swap} \\
\end{array}
\]

Pass n-1

\[
\begin{array}{cccccc}
1 & 3 & 5 & 6 & 7 & 8 \\
i \\
1 & 3 & 5 & 6 & 7 & 8 \\
\text{swap} \\
1 & 3 & 5 & 6 & 7 & 8 \\
\text{swap} \\
i,j \\
\end{array}
\]
Recursive Sort (MergeSort)

- Break sorting problem into smaller sorting problems and merge the results at the end
- MergeSort(0..n-1)
  - If list is size 1, return
  - Else
    - MergeSort(0..n/2)
    - MergeSort(n/2+1 .. n-1)
    - Combine each sorted list of n/2 elements into a sorted n-element list
Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels = 
    - \( \log_2(n) \)
  - Total operations to merge each level = 
    - \( n \) operations total to merge two lists over all recursive calls
- Mergesort = \( O(n \times \log(n)) \)
  - \( \log(n) \) is shorthand for \( \log_2(n) \) [i.e. log base 2]
Another Example

- Shown at the right are the binary combinations for different numbers of bits.
- Do you see a recursive pattern of the combinations as you look at progressively larger numbers of bits?
  - Hint: Start at the leftmost bit and move rightward.
Another Example

- If you are given the value, \( n \), and an array with \( n \) characters could you generate all the combinations of \( n \)-bit binary?
- Do so recursively!

http://cs103.usc.edu/websheets/index.php?#bincombos