Recursion, Part 2

Announcements

• This week’s lab is posted on the web.
• Test 1 will be returned in class.

Review from Friay’s Class

• Recursive functions are functions that make calls to themselves.
• Recursion works in implementation because each recursive call creates a separate instance of the recursive function, including parameters and local variables.
• Recursion is a powerful tool in problem solving. Recall the example of finding the number of $k$ element subsets of a set of size $n$.

Today’s Class

• Complete the discussion of Towers of Hanoi
• Recursion and linked-lists.
• Using recursion for “divide-and-conquer” algorithms:
  – Binary search
  – Merging and merge sort.

Recursive Definition of a Linked List

• We can define a linked-list recursively as
  – Empty, or
  – A head node that contains a pointer to a linked list
• The programming language Lisp was originally defined around this recursive definition.
• We can define recursive functions by thinking about the recursive
definition of a linked-list.

Recursive Linked-List Functions: Append

Here’s a recursive version of an append function.

• In fact, it uses two functions, one to find the last node in the list
and one to do the append. Assume the following node definition:

```cpp
class SNode {
public:
    SNode( const string& s ) : str(s), next(0) {};
    string str;
    SNode * next;
};
```

Also assume the linked list has a dummy head node.

• Here are the functions:

```cpp
SNode* find_last( SNode * head )
{
    if ( head->next )
        return find_last( head->next );
    else
        return head;
}

void append( SNode * head, const string& s )
{
    SNode* last = find_last( head );
    last -> next = new SNode( s );
}
```

• If `append` is a member function of a class that contains the head
pointer to a linked list as a member variable, then the original
call to `append` will not include `head`, but the call to `find_last`
STILL WILL.
Divide and Conquer Algorithms

We will start our discussion by considering the following problem:

You must guess a number that’s in the range 1 to 1000. When you guess a number you are told whether the correct number is greater or less (or equal) to the number you guessed? What strategy should you use to minimize the number of guesses?

Binary Search

Problem: given a SORTED array of values (increasing order), determine if $x$ is in the array.

- **Key insight:** If $x$ is less than the value stored in the middle of the array, then the search value must be in the lower half of the array. Otherwise it must be in the upper half.

- We can keep track of where we are searching using lower and upper search range subscripts.

```cpp
bool BinSearch( double arr[], int low, int high, double x )
{
    if ( low == high )
        return arr[low] == x;

    int mid = (low + high) / 2;
    if ( x <= arr[mid] )
        return BinSearch( arr, low, mid, x ); // lower half
    else
        return BinSearch( arr, mid+1, high, x ); // upper half
}
```

- If there are $n$ values in the array, the original call to the function will be

  `BinSearch( arr, 0, n-1, x );`

- We will explore this using the example:
Later in the semester we will see how to make this non-recursive and return the actual location containing the desired value.

Merging Two Sorted Arrays

This example is not a recursive function but will be used in a recursive sorting function below.

- **Problem:** Given two arrays sorted into increasing order, create a third array that includes the values from both arrays, in increasing order.

- **Example:** Merging

  \[
  \begin{array}{cccccc}
  0 & 1 & 2 & 3 & 4 \\
  3.2 & 5.7 & 13.7 & 31.8 & 33.5 \\
  \end{array}
  \]

  and

  \[
  \begin{array}{cccccc}
  0 & 1 & 2 & 3 & 4 \\
  8.9 & 10.4 & 12.2 & 17.4 & 23.9 \\
  \end{array}
  \]

  should create

  \[
  \begin{array}{ccccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  3.2 & 5.7 & 8.9 & 10.4 & 12.2 & 13.7 & 17.4 & 23.9 & 31.8 & 33.5 \\
  \end{array}
  \]

- **Key thoughts:** Where can the smallest value in the final array come from? Where can the second smallest value come from? We will use the answers to these questions to develop an intuitive understanding of the solution technique in class.

- The resulting code is attached on a separate page of this handout.
Merge Sort on Arrays

- Here’s an idea for sorting an array: split it in half, recursively sort each half, and merge the two resulting halves.
- Another way to think about this is to sort an interval within an array by splitting the interval in half, recursively sorting each half, and merging the resulting intervals.
- The base case is an interval of size 1.
- We already know how to do most of this:
  - In binary search we split an interval in two and searched one of the two pieces. Now we are going to work on each piece separately.
  - In the merge function, we merged two separate arrays, but these can also be two different parts of the same array.
  - The MergeSort code is attached to the handout.
  - We will go over examples in class.

Discussion of Merge Sort

- Merge sort would have been hard to design without recursion.
- The code is efficient except for the extra work of copying.
- The merging idea can also be used for linked lists. Here copying is not necessary:
  - Splitting requires breaking the linked list chain into two chains.
  - Merging requires joining the sorted subchains into a single sorted chain.