ECE 2574: Data Structures and Algorithms - Recursion II

C. L. Wyatt

Today we will look at two more examples of recursion and discuss performance issues

- Warmup
- Binary search of arrays
- Towers of Hanoi solution
- Efficiency of recursion in C++

Recursion is often useful when manipulating arrays.

Consider the binary search algorithm given a sorted array and a value to search for, the key.

- find the middle of the array
- if the key is in the middle slot, done
- if key is less, search just the lower half
- else search the upper half

Warmup #1

- Consider the array `{1,3,6,7,12,18,19}` and the binary search algorithm.
- Assume zero-based indexing.
- Suppose you are searching for a key $= 2$.

On the first call of the algorithm which index and value would you compare the key to? 69% correctly answered index 3, value 7.

A recursive version in pseudo-code

```
function search(data[], int lo, int hi, key) returns int

     if(lo > hi) return -1

     mid = floor( lo + (hi - lo) / 2)

     if (key < data[mid])
        return search(data, lo, mid-1, key)
     elseif (key > data[mid])
        return search(data, mid+1, hi, key)
     else
        return mid
endfunction
```

The first call is `mid = search(data, 0, length(data)-1, key)`.

A iterative version in pseudo-code

```
function search(data[], key) returns int

      int lo = 0
      int hi = length(data) - 1

      while (lo <= hi)
            mid = floor( lo + (hi - lo) / 2)
            if (key < data[mid])
                hi = mid-1
            elseif (key > data[mid])
                lo = mid+1
            else
                return mid
      endwhile

      return -1
endfunction
```

Question, why do either of these?

This implementation does not even require the array to be sorted.

```
function search(data[], key) returns int

     for(int i = 0; i < length(data); ++i)
          if(data[i] == key)
           return i
     endfor

     return -1
endfunction
```

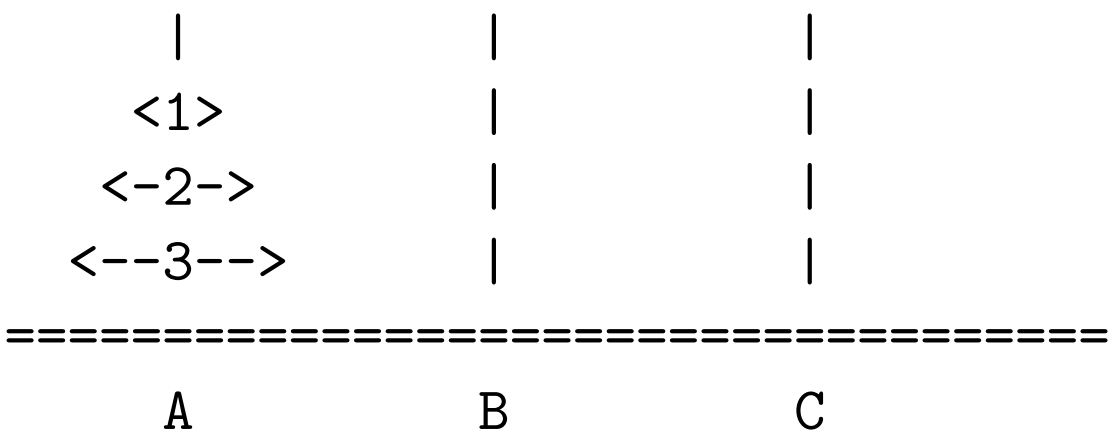
An important application of binary search is list filtering

- Credit-card numbers
- IP addresses
- Email sender filters
- on and on

These might be setup as either whitelist or blacklist filtering.

A classic recursion example: Towers of Hanoi

Move N disks from peg A to peg B, using C as an intermediary, at all times disks must be ordered largest to smallest vertically.



Recursive Solution to Towers of Hanoi (pseudo-code)

```
Function Towers(First, Aux, Last, n)
Input: Names of three pegs: First, Aux, Last
Output: solution to problem

  if( n == 1)
      write("Move disk 1 from peg" First "to Last)
  else
      Towers(First, Last, Aux, n-1)
      write("Move disk" n "from peg" First "to Last)
      Towers(Aux, First, Last, n-1)
  endif
endfunction
```

Warmup #2

The recurrence relation for the number of moves when solving the towers problem for n $>$ 0 disks, t(n) is
t(n) = 2*t(n-1) + 1; with I.C. t(1) = 1
or in closed form
t(n) = 2^n-1
How many moves does it take for n=4 disks versus n=10?
60% got this correct: 15 moves versus 1023 moves.

Number of moves required for Towers solution

Number of Disks n	Number of Moves t(n)
1	1
2	3
3	7
4	15
5	31
6	63
10	1023
100	1.2677 x 10^30
...	...

Warmup #3

What kind of recursion can be converted to a iterative algorithm?
80% correctly answered tail recursion.

Efficiency of recursion in practice

- many languages have much better support for recursion, e.g Haskell and Lisps
- we can simulate recursion using a stack (see lecture 13)
- In C and C++ it is hard to (portably) know how much stack you have used, but easier to track how much heap you have allocated

tail recursion

A tail-recursive function is one where

- there is a single recursive call
- it is the last statement in the recursive function

The simplest example for illustration

```
int fun(int x) {
   if ( x == 0 ) {
      return x;
   }
   return fun(x - 1);
}
```

Most C++ compilers can do tail-call optimization, effectively turning into an iterative procedure, when compiled with optimization flags.

Exercise: Binary Search

Lets implement the binary search algorithm operating on a std::array.

1. Download the starter code
2. In `search.hpp` implement the recursive binary search algorithm as defined.
3. In `search.hpp` implement the iterative binary search algorithm as defined.
4. Build your code locally as you work. Use the provided set of Catch tests.
5. Submit your `search.hpp` file via Canvas at the Assignment "Exercise for Meeting 6".

Next Actions and Reminders

- Read CH 95-111
- Warmup due by noon on Wednesday 9/13
- Project 1 will be released by Wed. It will be due Sat 9/23.