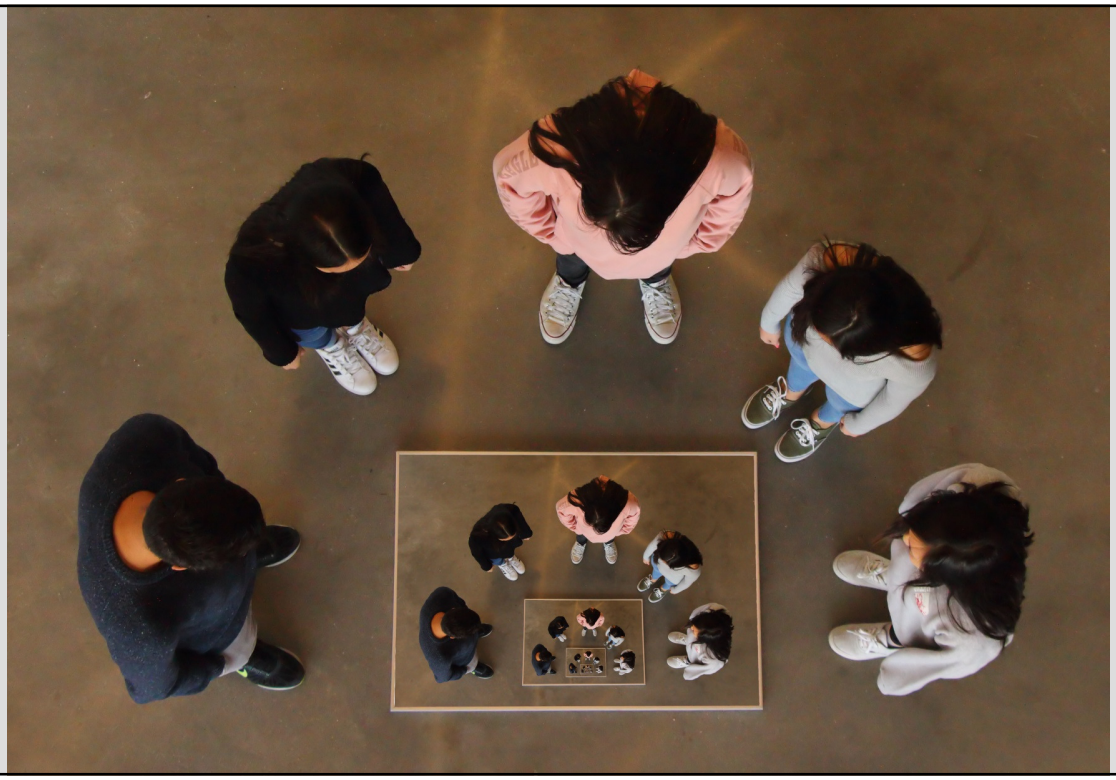
CMSC131

Recursion Part 2



Solving More Problems with Recursion

The examples we've looked at so far were useful for exploring the concept of and syntax for recursion.

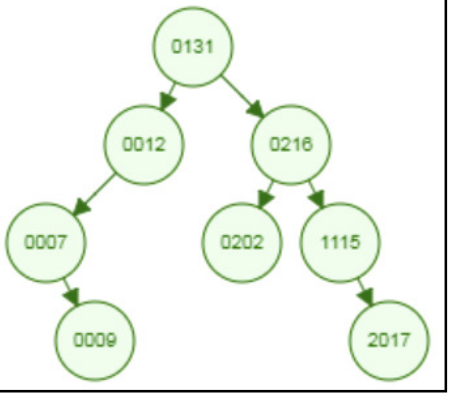
Let's now consider some problems whose solution is made much easier or efficient using recursion.

- Given a non-linear data structure, traverse it.
- Given a list of values, sort them.
- Given a list of values, figure out whether any subset of them add up to a target value.
- Find the greatest common denominator of two ints.

Trees

A tree is a data structure where you have individual nodes where each node has a single entry point, but can have multiple exit points.

How would you visit each node exactly once? For now let's assume there are at most two exits possible, **left** and **right**.



MergeSort

Thinking about the problem of sorting from a recursive point of view, we could describe it as:

- If there's just one element, it's sorted.
- Otherwise:
  - split the list in two parts (roughly half and half)
  - sort each of those two parts
  - merge those two now-sorted parts together

Being careful, we could do all of this within the array itself by creating two sub-lists logically…

https://www.youtube.com/watch?v=XaqR3G_NVoo

Pseudocode: MergeSort

```
MergeSort(
      int[] list,
      int start, int length
) {
  if the length is greater than 1
    find the "middle" of the region
    MergeSort the list from start to middle
    MergeSort the list from just past middle to end

    Merge the two now-sorted parts of the list
  }
}
```

Let's trace it with some examples…

Pseudocode: Merge

```
Merge (int[] array,
       int left_start, int left_size,
       int right_start, int right_size
) {
make a new mini-list "L" the size of the "left" list and copy
from that from the master array to "L" do the same for the
"right" list into "R"

set indices posL and posR to tops of "L" and "R"
while both of "L" and "R" have things in them, compare the "top"
things in each and put the smaller back into master array and
advance the "top" of the appropriate mini-list

copy the rest of whichever mini-list ("L" or "R") isn't empty yet
back into the master array
```

Big-O: MergeSort

One way to consider the Big-O runtime of this is that:

- a list of n values will be split roughly $\log_2 n$ times before getting down to at most a single element in every sublist (so $\log_2 n$ levels of recursion)
- every level's merges will need to roughly look at each value, so a cost of n per level

MergeSort is therefore in Big-O($n \log_2 n$)

- This will be shown in more detail in CMSC351.

Concurrency and MergeSort?

Imagine we had a computer with 8 processors that all shared the same memory bank (some of you might have such a computer).

How could MergeSort be customized to take advantage of this (assuming large input sizes) without too much additional code?

- What would the "speed-up" potential seem to be?
- Do you think there are other things that could stand in the way of this speed-up factor?

There are "real" problems (not contrived) whose optimal solution is still an open question.

Subset Sum: Problem Statement

Given a list of values and a target sum, can that target be made by adding a subset of the list of values?

NOTE: The "pure" version doesn't take in a target but rather sets the target to 0.

Example list: **3, 34, -7, 4, 12, 5, 2, 23**

Example target: **9**

Example target: **17**

Example target: **49**

The sum of all of the positive values is 83. Do you think that there are any target integer sums between 0 and 83 that can't be achieved? Try the first few numbers (0, 1, 2) by hand…

How many numbers between 0 and 83 **cannot** be made from subsets of: 3, 34, -7, 4, 12, 5, 2, 23

20% A. None

20% B. One

20% C. Two to Five

20% D. Five to Ten

20% E. More than Ten

Response Counter

Subset Sum: Possible Solution

Given a list of values and a target sum, can that target be made by adding a subset of the list of values? Let's consider a recursive way of thinking about the problem.

- If there's just one value in the list that is being considered then it comes down to whether or not that value is the target.
- Otherwise, there are two possibilities to consider when thinking recursively; using the first value in the list or not using it.

Subset Sum: recursion (page 1)

```
public static String
  targetSum(int startPoint, int[] values, int target) {

  //just one value in the list being considered
  if (startPoint==values.length-1) {
    if (values[startPoint]==target) {
      return Integer.toString(target);
    }
    else {
      return "";
    }
  }

//continued on next slide…
```

Subset Sum: recursion (page 2)

```
  //check possibility #1
  String withFirst = targetSum(
    startPoint+1, values, target-values[startPoint]);
  if (!withFirst.equals("")) {
    return values[startPoint]+", "+withFirst;
  }
  //if not, check possibility #2
  String withoutFirst = targetSum(
    startPoint+1, values, target);
  if (!withoutFirst.equals("")) {
    return withoutFirst;
  }
  //if you get here, neither possibility worked
  return "";
}
```

Big-O: Subset Sum

It turns out that this presented algorithm basically explores all possible subsets in the worst case and thus its worse case runtime is exponential in the input size.

We call this Big-O(2^n).

The question of whether or not there is a way to do this in a better runtime class is an open problem in computing (*and worth a million dollars*).

Greatest Common Denominator

The following works and is very efficient but is it obvious why it gives the correct answer?

```
public static int doGCD (int n1, int n2)
{
  //put in ascending order if not already
  if (n2<n1) return doGCD(n2,n1);

  //look for stopping case
  if ( (n2%n1)==0 ) return n1;

  //recursive call on subproblem
  return doGCD(n2%n1, n1);
}
```

"Towers" problems

Given 3 pegs and 6 disks of different sizes in ascending size on peg 1, move them to peg 3 (using peg 2 as needed) one disk at a time so that no larger disk is ever above a smaller one.

This is a smaller version of what's called the "Towers of Brahma" problem (64 disks).

A more generic version is Tower(nDisks, nPegs).

Tail Recursion

In some programming languages, if the recursive call is the last thing to be executed in the recursive method, the compiler can utilize optimizations to better use memory, etc.

- It needs to be the very last thing, which is not the same as being on the last line of code.
- Java does NOT optimize tail recursive code currently…

Calculating the n^{th} Fibonacci

Fib(0) = 0, Fib(1) = 1, Fib(n)=Fib(n-1)+Fib(n-2)

Does this lend itself well to a recursive solution?

What about tail recursion?

Regular Recursion (not tail)

```
public static long fibV1(int n) {
   if (n <= 1)  return n;
   return fibV1(n-1)+fibV1(n-2);
}
```

Tail Recursion (but also other "trickery")

```
public static long fibV2(long n, long a, long b) {
   if (n == 0)  return a;
   if (n == 1)  return b;
   return fibV2(n-1, b, a + b);
}

//NOTE: initial call needs to be fibV2(#, 0, 1);
```