Recursion

- Arises in two forms in computer science
- We’ll explore both
  - Recursion as a mathematical tool for defining a function in terms of its own value in a simpler case
  - Recursion as a programming tool. You’ve seen this previously but we’ll take it to mind-bending extremes (by the end of the class it will seem easy!)

Recursion as a math technique

- Broadly, recursion is a powerful technique for specifying functions, sets, and programs
- Example recursively-defined functions and programs
  - factorial
  - combinations
  - exponentiation (raising to an integer power)
- Example recursively-defined sets
  - grammars
  - expressions
  - data structures (lists, trees, ...)

The Factorial Function (n!)

- Define n! = n·(n-1)·(n-2)···3·2·1  read: “n factorial”
  - E.g., 3! = 3·2·1 = 6
  - By convention, 0! = 1
- The function int \(\rightarrow\) int that gives n! on input n is called the factorial function

The Factorial Function (n!)

- n! is the number of permutations of n distinct objects
  - There is just one permutation of one object. 1! = 1
  - There are two permutations of two objects: 2! = 2
    - 1 2
    - 2 1
  - There are six permutations of three objects: 3! = 6
    - 1 2 3
    - 1 3 2
    - 2 1 3
    - 2 3 1
    - 3 1 2
    - 3 2 1
  - If n > 0, n! = n·(n - 1)!

Permutations of non-orange blocks

- Total number = 4·3! = 4·6 = 24: 4!

- Each permutation of the three non-orange blocks gives four permutations when the orange block is included
Observation

One way to think about the task of permuting the four colored blocks was to start by computing all permutations of three blocks, then finding all ways to add a fourth block.

- And this “explains” why the number of permutations turns out to be 4!
- Can generalize to prove that the number of permutations of n blocks is n!

A Recursive Program

\[ 0! = 1 \]
\[ n! = n \cdot (n-1)! \quad n > 0 \]

\[
\text{static int fact(int n) }
\begin{align*}
&\text{if } (n == 0) \text{ return 1;} \\
&\text{else return } n \times \text{fact}(n-1);
\end{align*}
\]

General Approach to Writing Recursive Functions

1. Try to find a parameter, say n, such that the solution for n can be obtained by combining solutions to the same problem using smaller values of n (e.g., (n-1) in our factorial example).
2. Find base case(s) – small values of n for which you can just write down the solution (e.g., 0! = 1).
3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases.

A cautionary note

- Keep in mind that each instance of your recursive function has its own local variables.
- Also, remember that “higher” instances are waiting while “lower” instances run.
- Not such a good idea to touch global variables from within recursive functions.

The Fibonacci Function

- Mathematical definition:
  \[ \text{fib}(0) = 0 \]
  \[ \text{fib}(1) = 1 \]
  \[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \quad n \geq 2 \]

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, ...

\[
\text{static int fib(int n) }
\begin{align*}
&\text{if } (n == 0) \text{ return 0;} \\
&\text{else if } (n == 1) \text{ return 1;} \\
&\text{else return fib(n-1) + fib(n-2);} \\
&\text{fib}(n-1) + \text{fib}(n-2); \\
&\text{fib}(n-2) \quad \text{fib}(n-1)
\end{align*}
\]

Recursive Execution
One thing to notice

- This way of computing the Fibonacci function is elegant, but inefficient
- It "recomputes" answers again and again!
- To improve speed, need to save known answers in a table!
- Called a cache

Adding caching to our solution

Before:

```java
static int fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

After:

```java
ArrayList<boolean> known = new ArrayList<boolean>;
ArrayList<int> cached = new ArrayList<cached>;

static int fib(int n) {
    v = 0;
    if (known[n])
        return v;
    else
        return fib(n-1) + fib(n-2);
}
```

Notice the development process

- We started with the idea of recursion
- Created a very simple recursive procedure
- Noticed it will be slow, because it wastefully recomputes the same thing again and again
- So made it a bit more complex but gained a lot of speed in doing so
- This is a common software engineering pattern

Combinations (a.k.a. Binomial Coefficients)

- How many ways can you choose r items from a set of n distinct elements? \( \binom{n}{r} \) "n choose r" 
  \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \)
- 2-element subsets containing A: \( \binom{4}{1} \)
- 2-element subsets containing A: \( \binom{4}{2} \)
- Therefore, \( \binom{5}{2} = \binom{4}{1} + \binom{4}{2} \)
- ... in perfect form to write a recursive function!

Combinations

\[
\begin{align*}
\binom{0}{0} &= 1 \\
\binom{n}{0} &= 1 \\
\binom{n}{1} &= n \\
\binom{n}{r} &= \binom{n-1}{r-1} + \binom{n-1}{r} \\
\end{align*}
\]

Can also show that \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \)

Pascal’s triangle

\[
\begin{array}{ccccccc}
1 & & & & & & \\
1 & 1 & & & & & \\
1 & 2 & 1 & & & & \\
1 & 3 & 3 & 1 & & & \\
1 & 4 & 6 & 4 & 1 & & \\
\end{array}
\]

Binomial Coefficients

Combinations are also called binomial coefficients because they appear as coefficients in the expansion of the binomial power \( (x+y)^n \): 

\[
(x+y)^n = \binom{n}{0} x^n y^0 + \binom{n}{1} x^{n-1} y + \binom{n}{2} x^{n-2} y^2 + \cdots + \binom{n}{n} y^n \\
= \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i
\]
Combinations Have Two Base Cases

$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}, \quad n > r > 0$

$\binom{n}{0} = 1$

$\binom{n}{n} = 1$

Two base cases

Coming up with right base cases can be tricky!

General idea:
- Determine argument values for which recursive case does not apply
- Introduce a base case for each one of these

Recursive Program for Combinations

$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}, \quad n > r > 0$

$\binom{n}{0} = 1$

$\binom{0}{r} = 1$

```java
static int combs(int n, int r) {   //assume n>=r>=0
    if (r == 0 || r == n) return 1; //base cases
    else return combs(n-1,r) + combs(n-1,r-1);
}
```

Exercise for the reader (you!)

Modify our recursive program so that it caches results

Same idea as for our caching version of the fibonacci series

Question to ponder: When is it worthwhile to adding caching to a recursive function?
- Certainly not always...
- Must think about tradeoffs: space to maintain the cached results vs speedup obtained by having them

Positive Integer Powers

$a^n = a \cdot a \cdot a \cdots a$ (n times)

Alternate description:
- $a^0 = 1$
- $a^{n+1} = a \cdot a^n$

```java
static int power(int a, int n) {
    if (n == 0) return 1;
    else return a*power(a,n-1);
}
```

A Smarter Version

Power computation:
- $a^1 = 1$
- If n is nonzero and even, $a^n = (a^{n/2})^2$
- If n is odd, $a^n = a \cdot (a^{n/2})^2$

Java note: If x and y are integers, "x/y" returns the integer part of the quotient

Example:
$a^5 = a \cdot (a^2)^2 = a \cdot (a^2)^2 = a \cdot (a^2)^2$

Note: this requires 3 multiplications rather than 5!

What if $n$ were larger?
- Savings would be more significant
- Much faster than the straightforward computation

Smarter computation: $\log(n)$ multiplications

```java
parameters

A Smarter Version in Java

n = 0: $a^0 = 1$
- n nonzero and even: $a^n = (a^{n/2})^2$
- n nonzero and odd: $a^n = a \cdot (a^{n/2})^2$

local variable

```java
static int power(int a, int n) {
    if (n == 0) return 1;
    int halfPower = power(a,n/2);
    if (n%2 == 0) return halfPower*halfPower;
    return halfPower*halfPower*a;
}
```

- The method has two parameters and a local variable
- Why aren’t these overwritten on recursive calls?
Implementation of Recursive Methods

Key idea:
- Use a stack to remember parameters and local variables across recursive calls
- Each method invocation gets its own stack frame

A stack frame contains storage for
- Local variables of method
- Parameters of method
- Return info (return address and return value)
- Perhaps other bookkeeping info

Stack Frame

- A new stack frame is pushed with each recursive call
- The stack frame is popped when the method returns
- Leaving a return value (if there is one) on top of the stack

How Do We Keep Track?

At any point in execution, many invocations of power may be in existence
- Many stack frames (all for power) may be in Stack
- Thus there may be several different versions of the variables a and n

How does processor know which location is relevant at a given point in the computation?

Example: power(2, 5)

FBR

Computational activity takes place only in the topmost (most recently pushed) stack frame
## Conclusion

- Recursion is a convenient and powerful way to define functions.

- Problems that seem insurmountable can often be solved in a “divide-and-conquer” fashion:
  - Reduce a big problem to smaller problems of the same kind, solve the smaller problems.
  - Recombine the solutions to smaller problems to form solution for big problem.

- Important application (next lecture): parsing.