CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Spring 2012

Doug James

Lecture 5: Recursion

Visual Recursion

http://serendip.brynmawr.edu/exchange/files/authors/faculty/39/literarykinds/infinite_mirror.jpg

Recursion Overview

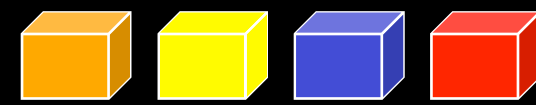
- Recursion is a powerful technique for specifying functions, sets, and programs
- Example recursively-defined functions and programs
  - factorial
  - combinations
  - exponentiation (raising to an integer power)
  - solution of combinatorial problems (i.e. search)
- Example recursively-defined sets
  - grammars
  - expressions
  - data structures (lists, trees, ...)

The Factorial Function (n!)

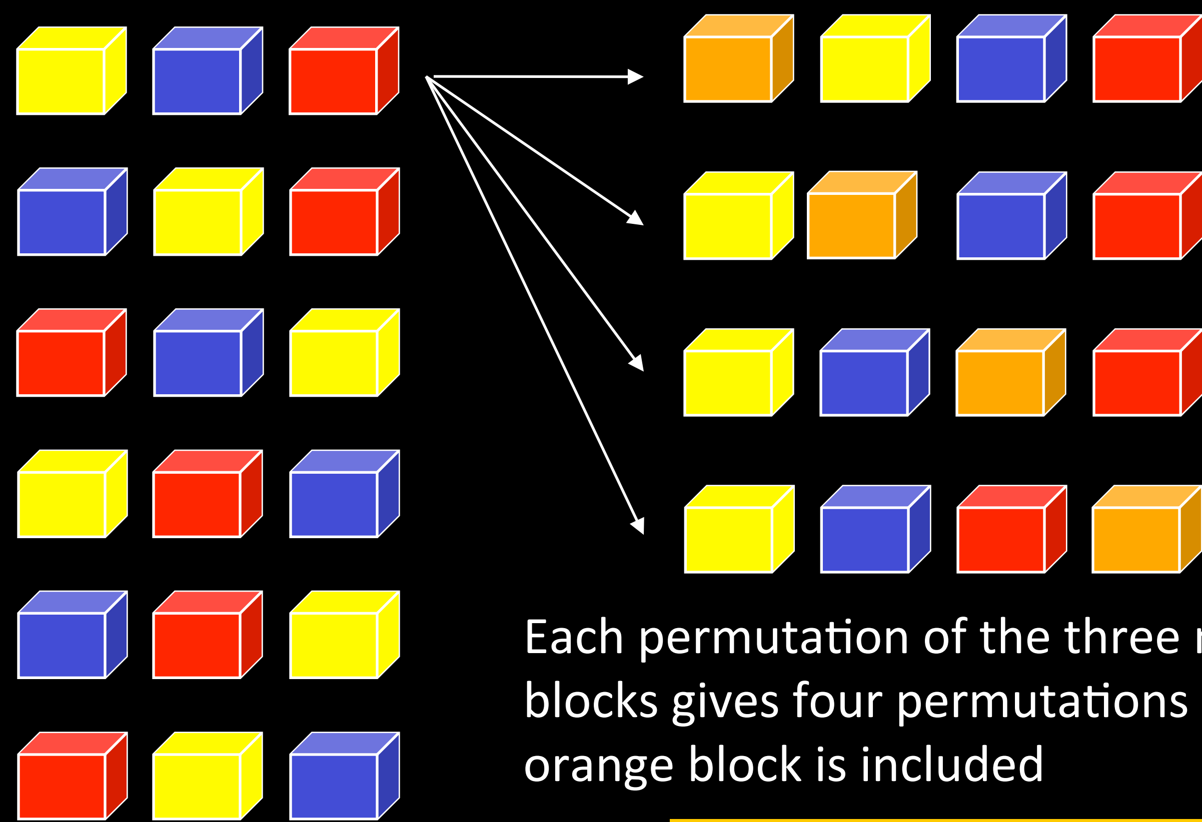
- Define: $n! = n\cdot(n-1)\cdot(n-2)\cdots 3\cdot 2\cdot 1$
  - read: "n factorial"
  - E.g., $3! = 3\cdot 2\cdot 1 = 6$
- The function int ➔ int that gives n! on input n is called the factorial function
- n! is the number of permutations of n distinct objects
  - There is just one permutation of one 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

Permutations of 

Permutations of non-orange blocks



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

Total number = $4 \cdot 6 = 24 = 4!$

→ General:

- $0! = 1$ (by convention)
- If $n > 0$, $n! = n \cdot (n-1)!$

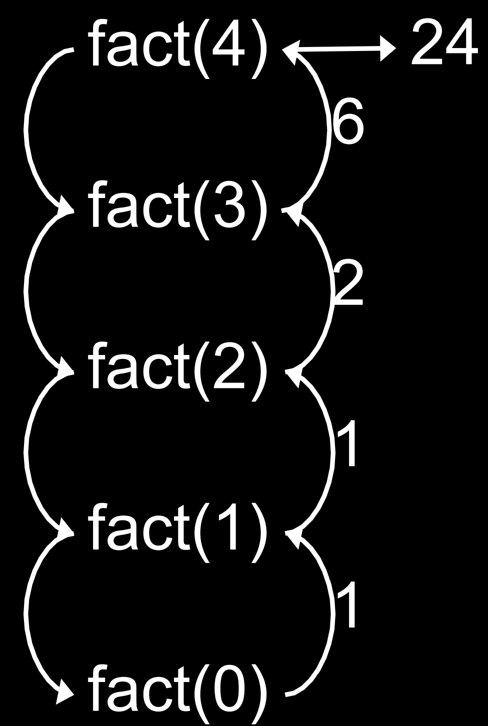
A Recursive Program

Recursive definition of n!

- 0! = 1
- n! = n·(n-1)!, n > 0

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n*fact(n-1);
}
```

Execution of fact(4)



General Approach to Writing Recursive Functions

- 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)!) (i.e. recursion)
- Find base case(s) – small values of n for which you can just write down the solution (e.g., 0! = 1)
- Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

The Fibonacci Function

- Mathematical definition:
  fib(0) = 0
  fib(1) = 1
  fib(n) = fib(n − 1) + fib(n − 2), n ≥ 2

  two base cases!

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

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

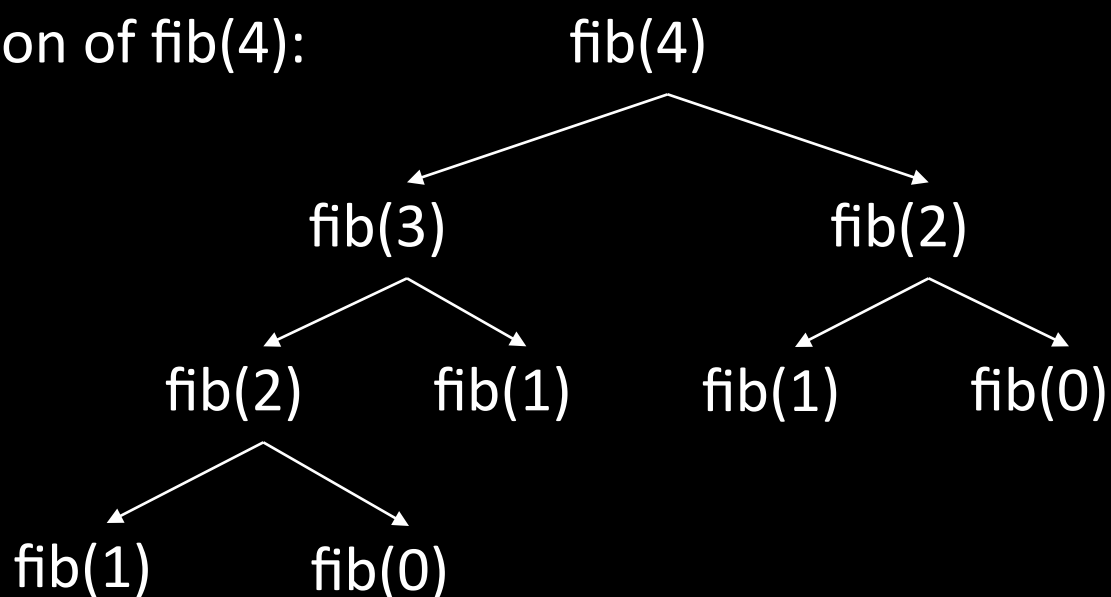
Fibonacci (Leonardo Pisano) 1170-1240?

Statue in Pisa, Italy, Giovanni Paganucci, 1863

Recursive Execution

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

Execution of fib(4):



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"
  - $\binom{5}{2}$ = number of 2-element subsets of {A,B,C,D,E}
    - 2-element subsets containing A: $\binom{4}{1}$
      {A,B}, {A,C}, {A,D}, {A,E}
    - 2-element subsets not containing A: $\binom{4}{2}$ {B,C},{B,D},{B,E}, {C,D},{C,E},{D,E}

- Therefore, $\binom{5}{2} = \binom{4}{1} + \binom{4}{2}$

Combinations

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

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

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

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

$$\binom{0}{0}$$
$$\binom{1}{0} \quad \binom{1}{1}$$
$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$
$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$
$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

Pascal's triangle

=

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Binomial Coefficients

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

$$(x + y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n}y^n$$

Multiple Base Cases

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

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

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

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

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

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

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

```
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);
}
```

Positive Integer Powers

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

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

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

A Smarter Version

- Power computation:
  - $a^0 = 1$
  - If n is nonzero and even, $a^n = (a^{n/2})^2$
  - If n is odd, an = $a \cdot (a^{n/2})^2$
    - Java note: If x and y are integers, "x/y" returns the integer part of the quotient

- Example:
  - $a^5 = a \cdot (a^{4/2})^2 = a \cdot (a^2)^2 = a \cdot ((a^{2/2})^2)^2 = a \cdot (a^2)^2$
  - Note: this requires 3 multiplications rather than 5!

- What if n were larger?
  - Savings would be more significant
  - Straightforward computation: n multiplications
  - Smarter computation: log(n) multiplications

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

parameters

```
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;
}
```

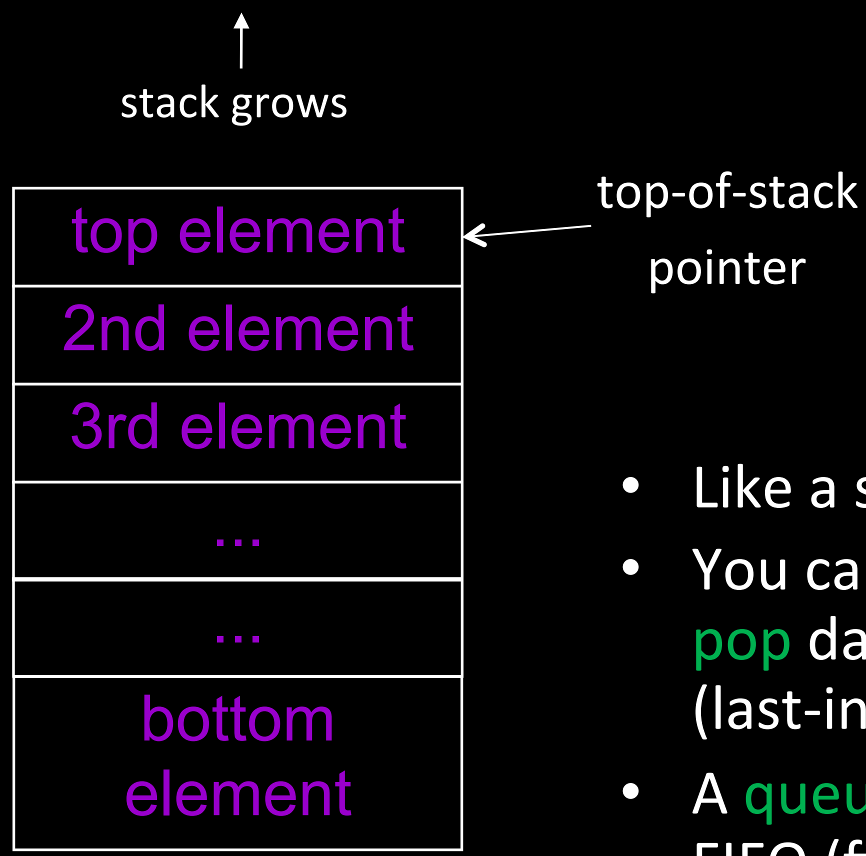
- The method has two parameters and a local variable
- Why aren't these overwritten on recursive calls?

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

Stacks

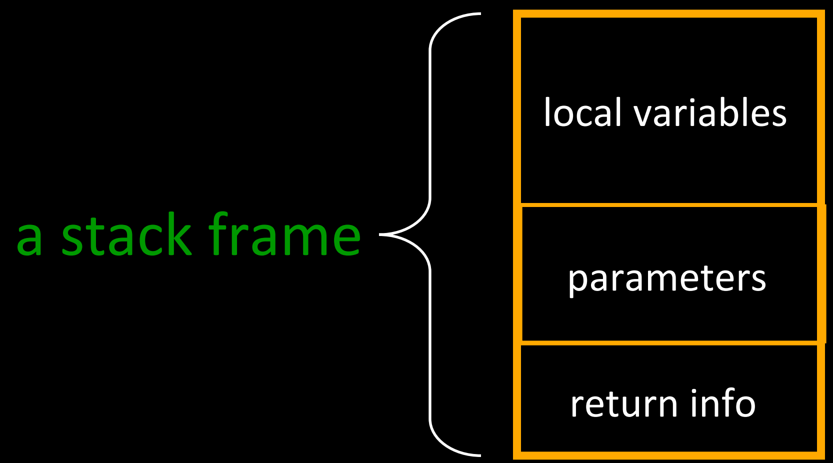


- Like a stack of plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

Stack Frame

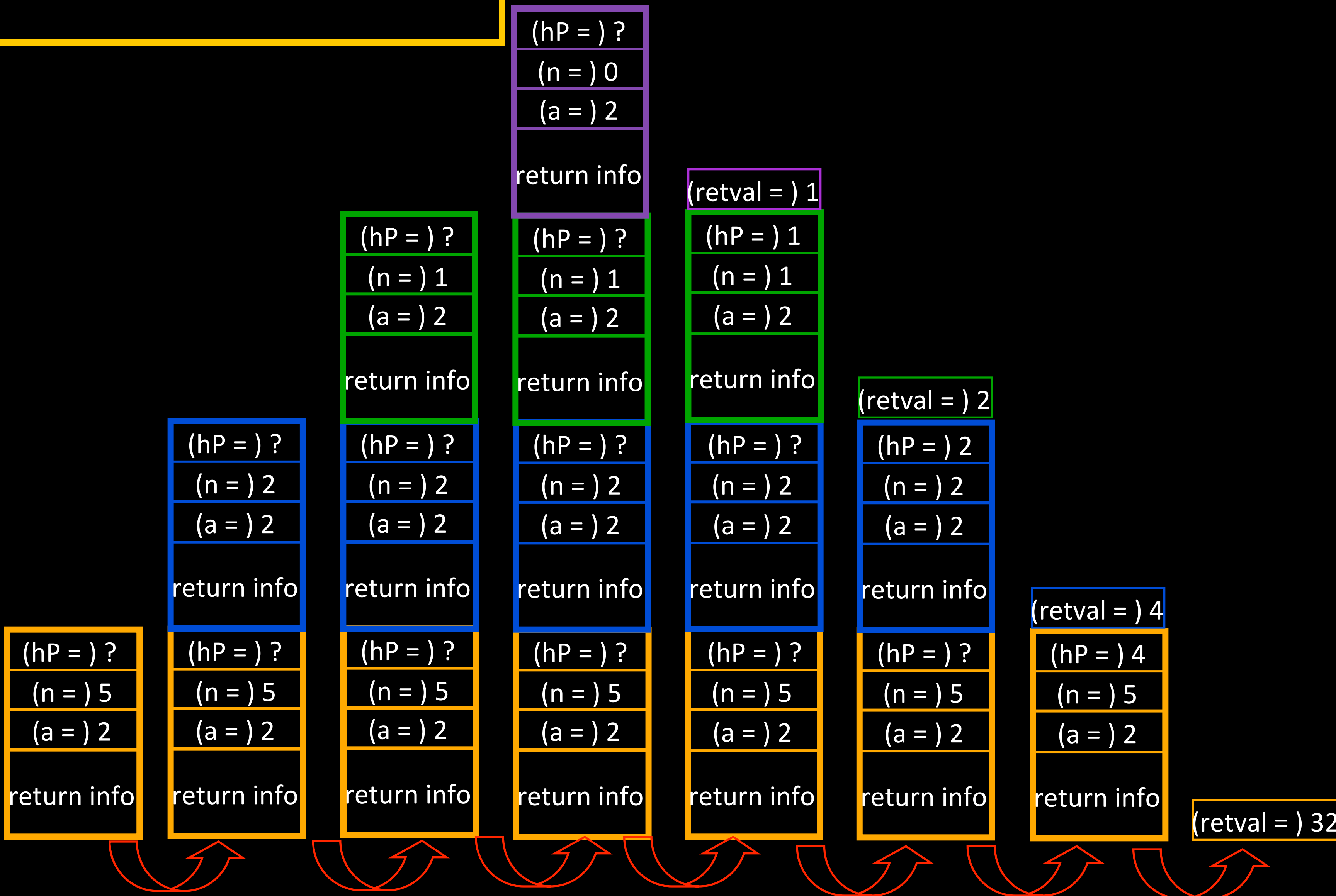
- A new stack frame is pushed with each recursive call

- The stack frame is popped when the method returns
  → Leaving a return value (if there is one) on top of the stack



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

Example: power(2, 5)



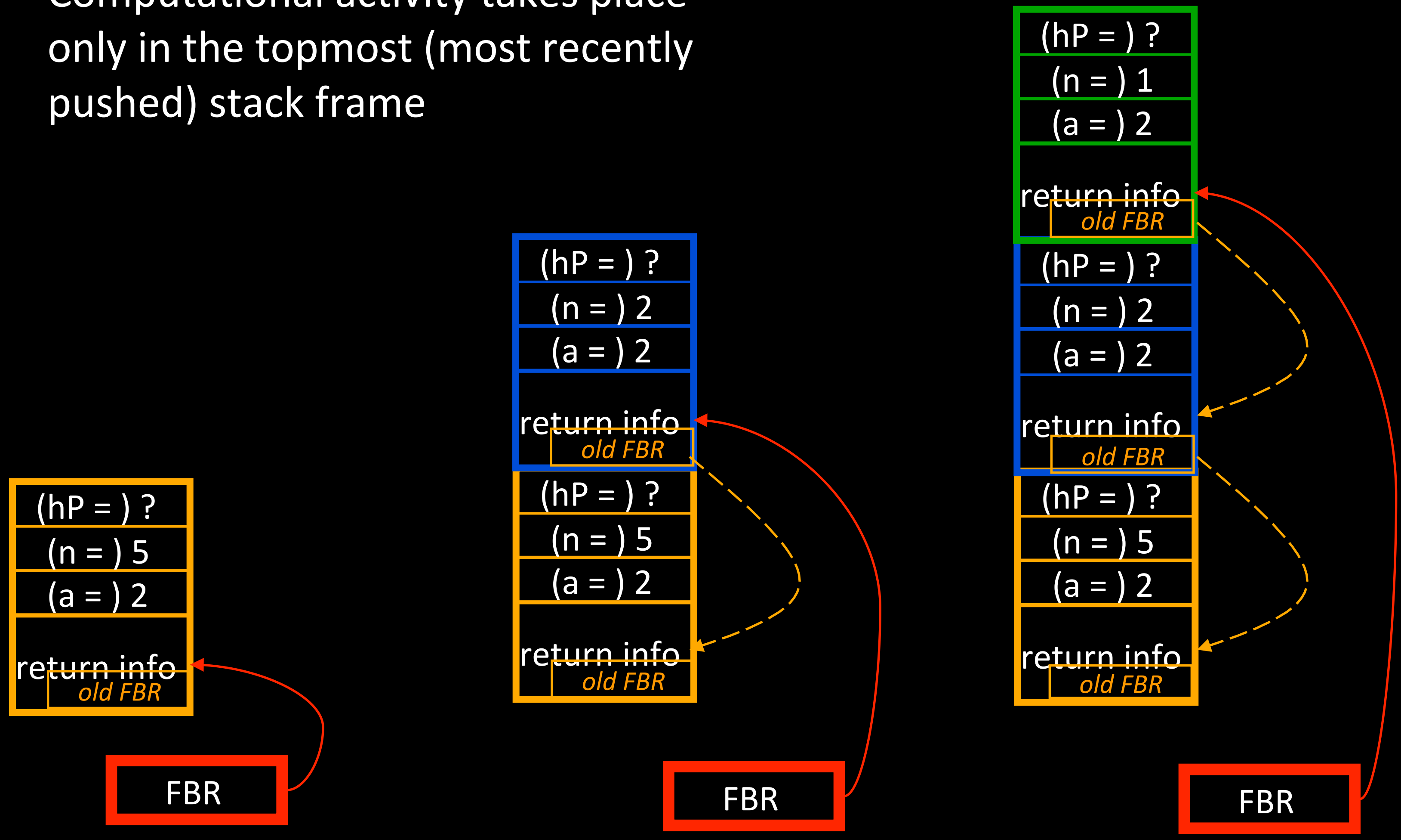
How Do We Keep Track?

- 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?
  → Frame Base Register
  - When a method is invoked, a frame is created for that method invocation, and FBR is set to point to that frame
  - When the invocation returns, FBR is restored to what it was before the invocation
- How does machine know what value to restore in the FBR?
  - This is part of the return info in the stack frame

FBR

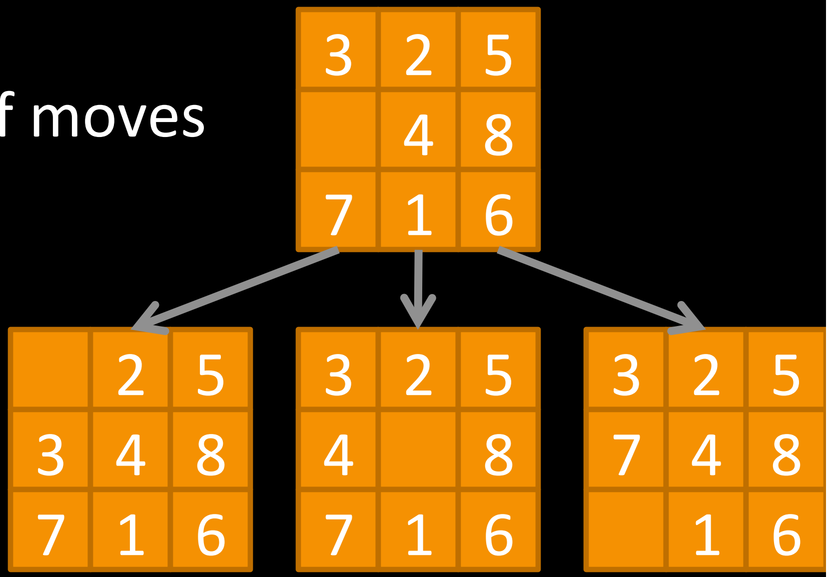
- Computational activity takes place only in the topmost (most recently pushed) stack frame



Problem Solving by Search

- Idea: Try all possible sequences of moves
- Pseudocode:
  - DepthFirstSearch(state)

```
IF isSolution(state) THEN
    RETURN(true)
WHILE hasNextLegalMove(state)
    next= getNextLegalMove(state)
    IF DepthFirstSearch(next) THEN
        RETURN(true)
RETURN(false)
```



- Caution: You might get a program that does not terminate, if you have
  - move sequences that can be infinitely long
  - move sequences that get you back to the same state (cycles)

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 applications:
  - Parsing (next lecture)
  - Collision detection