- Recursion
  - a programming strategy for solving large problems
  - Think “divide and conquer”
  - Solve large problem by splitting into smaller problems of same kind
- Induction
  - A mathematical strategy for proving statements about large sets of things
- Now we learn recursion…

# Recursion

Recursive definition[1]: See *recursive definition*.

Weiss: ch 7.0-5

[1]Free On-line Dictionary of Computing (FOLDOC)

# Recursive Definitions (recap)

| | | |
|---|---|---|
| $S(n) = S(n-1) + n$ | $S(0) = 0$ | $0 + 1 + \ldots + n$ |
| $Q(n) = Q(n-1) + n^2$ | $Q(1) = 1$ | $1^2 + 2^2 + \ldots + n^2$ |
| Tile($2^N$ by $2^N$) :<br>Place "L" in center;<br>Then tile each quadrant | Tile(2 by 2) :<br>easy | Tile entire grid<br>minus one square |
| $C(n) = C(n - 1) + 1$ | $C(0) = 5$ | |

# Recursion in Detail: Permutations

- How many ways to put 4 ducks in a row?

(4 very unique ducks)

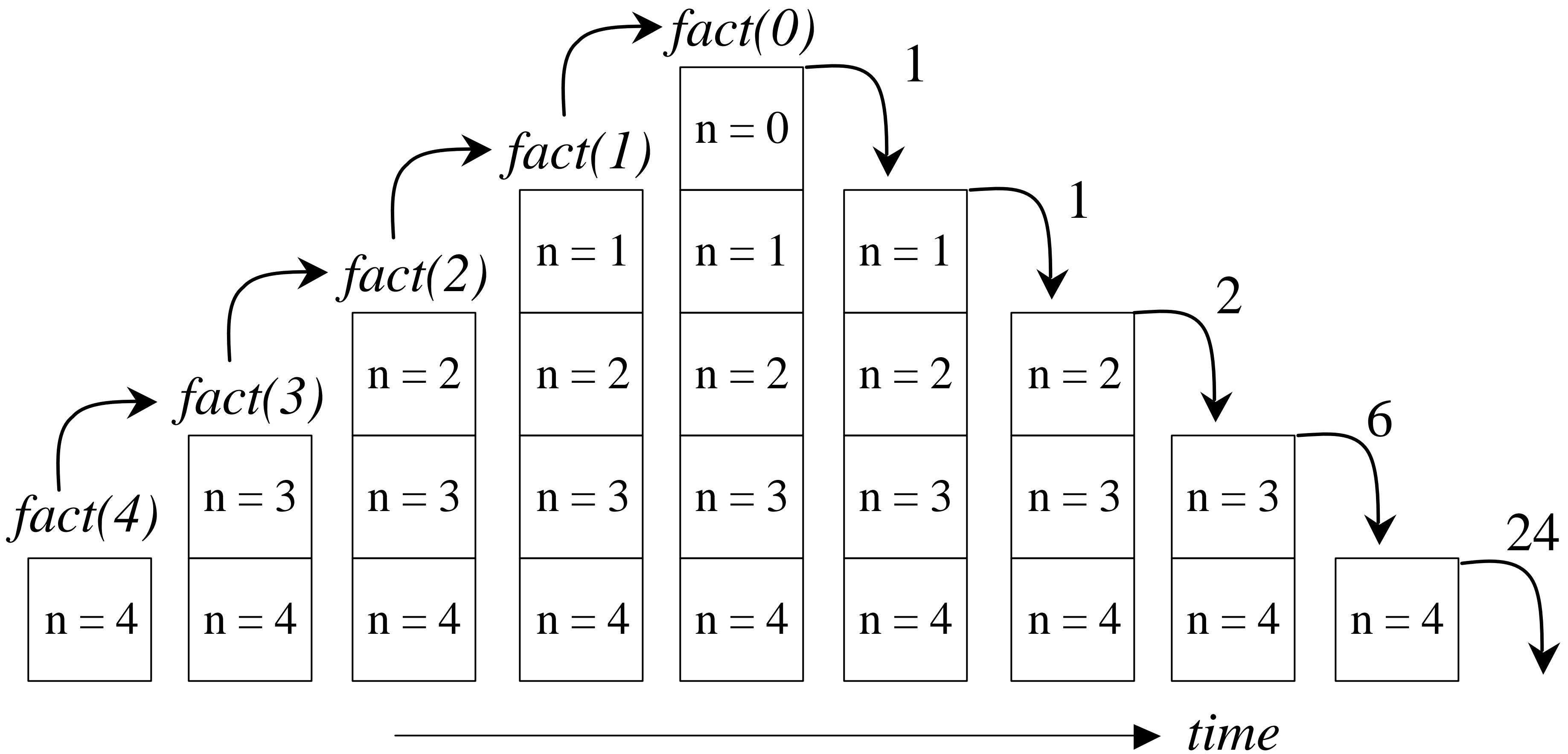
# Factorial

- How many ways to arrange $n$ distinct objects?
  - Call it *fact(n)*
  - Write it $n!$
  - Say it "*n factorial*"
- One definition (closed form):
  - $fact(n) = 1*2*3*...*n = n!$
- Another (recursive form):
  - $fact(1) = 1$
  - $fact(n) = fact(n-1) * n$, *for* $n > 1$
- By convention:
  - $fact(0) = 1$

# Code

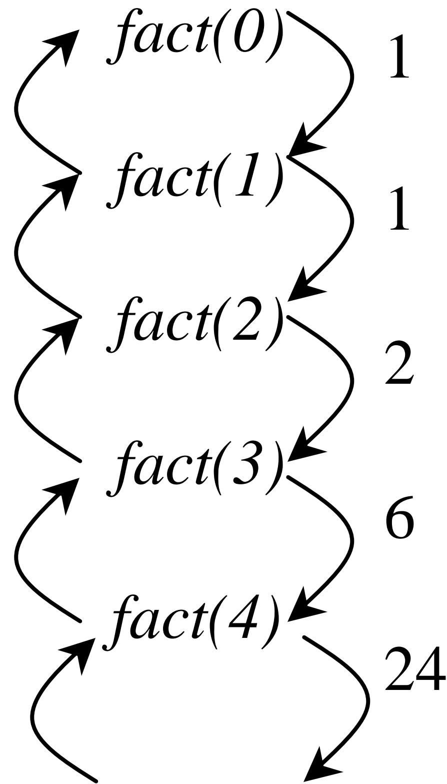
```
static int fact(int n) {
        if (n = = 0) return 1;
        else return fact(n –1) * n;
}
```

# Executing *fact(4)*



# Executing *fact(4)*

- Alternative view: *call tree*



# Writing Recursive Functions

1. Find a parameter $n$ such that solution for large $n$ can be obtained from solution(s) for smaller $n$.
2. Find base cases: small values of $n$ which you can solve directly, easily.
3. Verify that for any $n$, it will eventually reduce to a base case (or cases).
4. Write code.

# Fibonacci

- A problem in the third section of *Liber abaci* led to the introduction of the Fibonacci numbers and the Fibonacci sequence for which Fibonacci is best remembered today[1]:
  - *A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?*

[1]Taken from http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Fibonacci.html

# Fibonacci

- Definition (recursive form):

$$fib(0) = 1 \quad fib(1) = 1 \quad \text{Base cases}$$

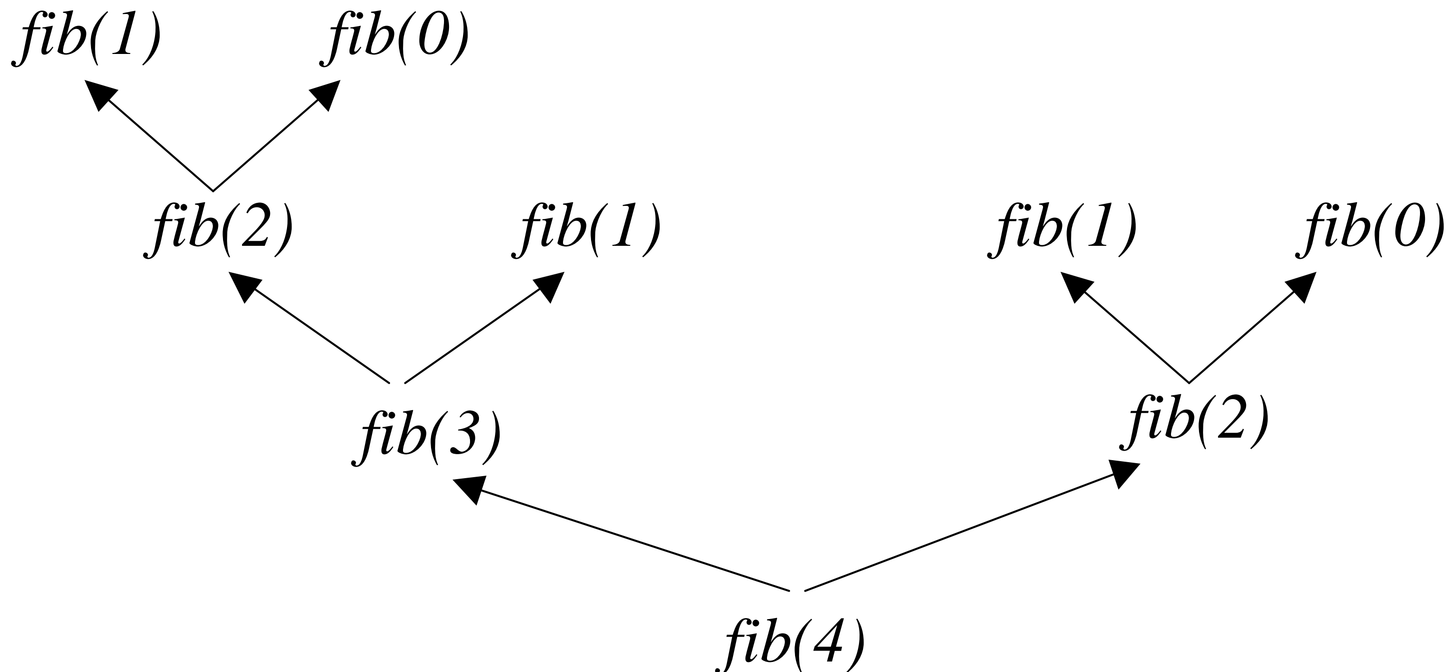
$$fib(n) = fib(n\text{-}1) + fib(n\text{-}2)$$

- Closed form? See HW.
- Example:

  1, 1, 2, 3, 5, 8, 13, 21, …
- Code:

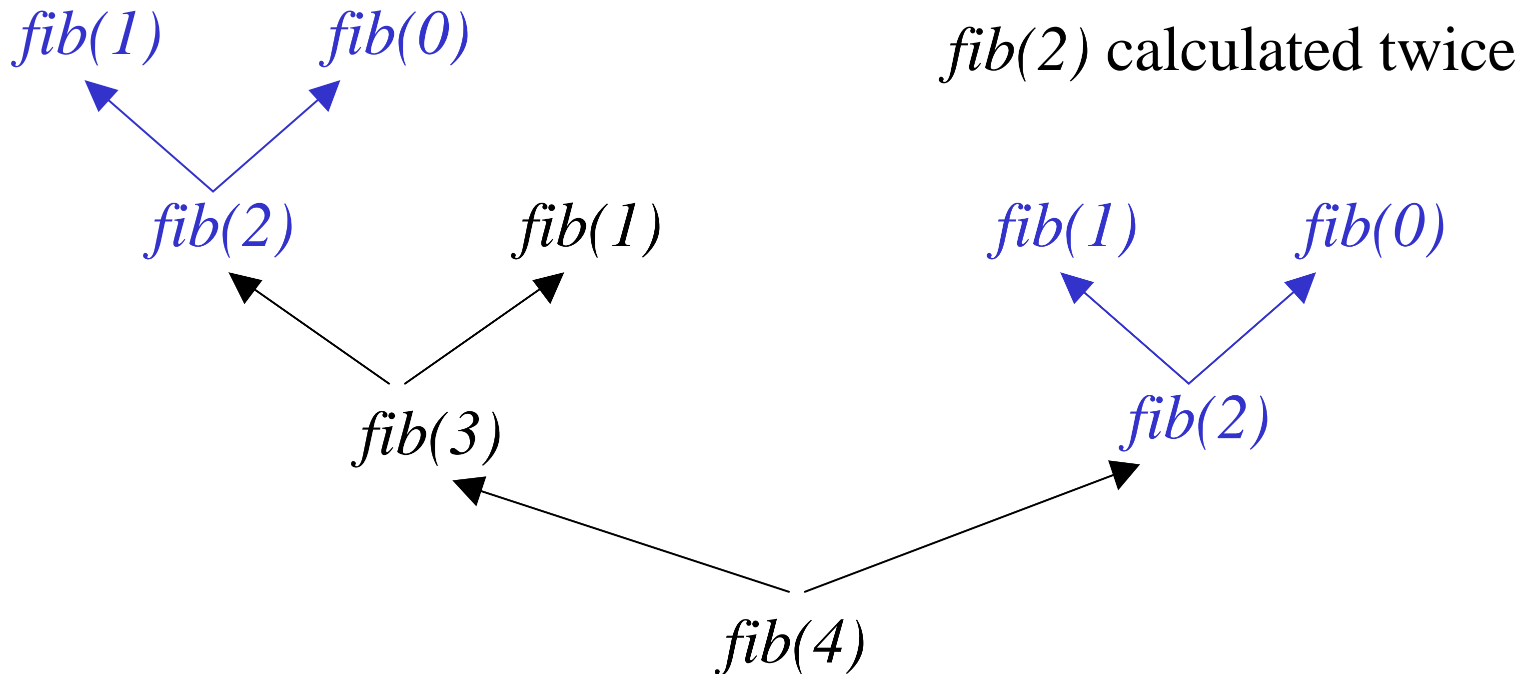
```
static int fib(int n) {
        if (n == 0 || n == 1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

# Call Tree for *fib(4)*




Note: this is called a *tree* because it has a *root,* some *leaves*, and it *branches*

# Inefficient!



# A More Complex Example: Combinations

- How many ways can you pick *r* items from a set of *n* distinct elements?
  - Call it ${}^{n}C_{r}$
- Ex. Pick two letters from set

  S = {A, B, C, D, E}

u

# A More Complex Example: Combinations

- How many ways can you pick *r* items from a set of *n* distinct elements?
  - Call it ${}^{n}C_{r}$
- Ex. Pick two letters from set

  S = {A, B, C, D, E}
- Answer:

  {A, B}, {B, C}, {B, D}, {B, E}

  {A, C}, {A, D}, {A, E}, {C, D}, {C, E}, {D, E}

  So 10 ways to choose.
- Recurrence relation?

# Combinations

- Theorem:

  ${}^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1}$ for $n > r > 0$ (recursive case)

  ${}^{n}C_{n} = 1$ (base case)

  ${}^{n}C_{0} = 1$ (base case)

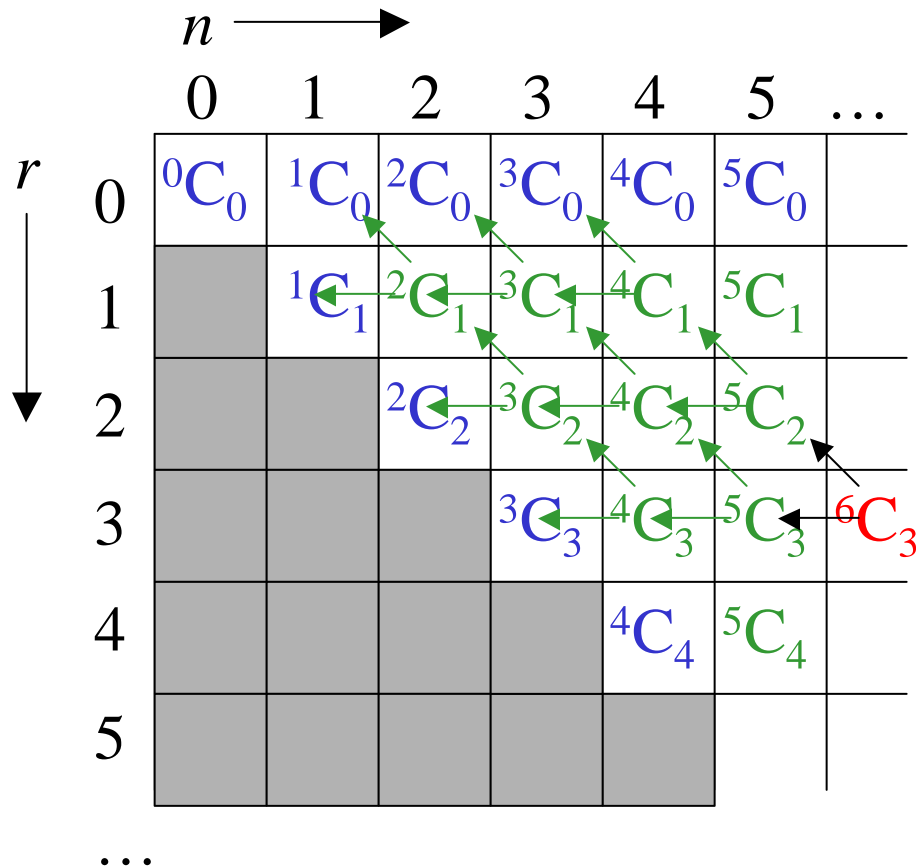
- Proof: (reason it out in three cases)
- Theorem:

  ${}^{n}C_{r} = n! / r!(n-r)!$

- Proof: (induction, or reason it out)

# Sanity Check

- Do all cases reduce to base cases?
- Example: ${}^6C_3 = {}^5C_3 + {}^5C_2 = \ldots = ?$



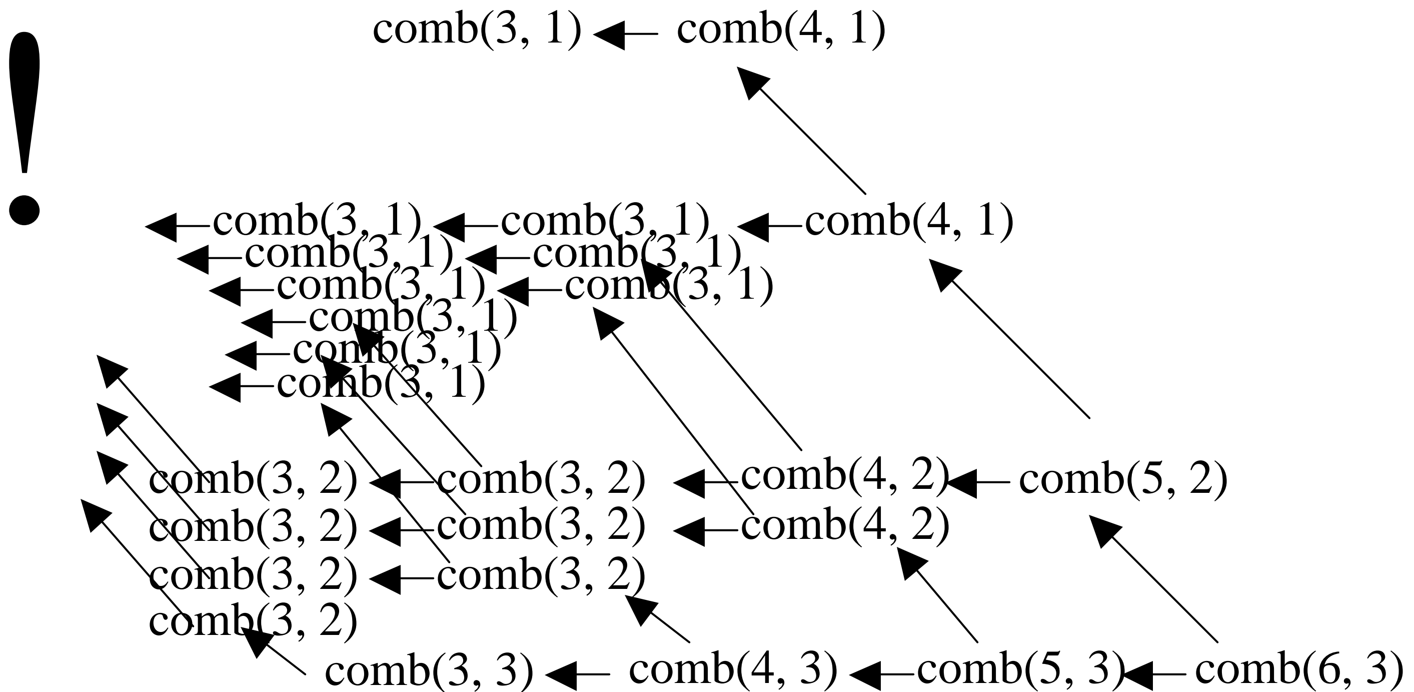

note: we call this a *graph*

# Code

```
static int comb(int n, int r) {
    if (r == 0 || n == r) return 1;
    else return comb(n-1, r) + comb(n-1, r-1);
}
```

# Somewhat inefficient code…

- Call tree:



# Efficiency

- Rule #1: Avoid duplicate work in recursive calls
  - e.g. iterative version of *fib(n)* is faster than recursive
- Other goals:
  - choose sub-problems carefully

    e.g. divide and conquer works best when we "divide" roughly evenly:

    16 → 8 → 4 → 2 → 1

    versus

    16 → 15 → 14 → 13 → 12 → 11 → 10… → 3 → 2 → 1
  - choose base cases carefully

    e.g., can stop recursion early for special cases

# Case Study

- Computing powers
  - pow(x, n) = $x * x * \ldots * x = x^n$

# First Attempt

- Recursive form:

  $x^0 = 1$

  $x^n = x^{n-1} * x$, for $n > 0$

- Code

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

# Efficiency

- Duplicate work in recursive calls?
- How much “work”?
  - Number of calls to pow()?
  - Number of multiplications?

# Second Attempt

- Recursive form:

  $x^0 = 1$

  $x^n = (x^{n/2})^2$, for $n > 0$, $n$ even

  $x^n = x*(x^{n/2})^2$, for $n > 0$, $n$ odd

- Code

```
static int pow(int x, int n) {
    if (n == 0) return 1;
    int halfPower = pow(x, n/2);
    if (n/2 == n) return halfPower * halfPower;
    else return x * halfPower * halfPower;
}
```

# Efficiency

- Duplicate work in recursive calls?
- How much “work”?
  - Number of calls to pow()?
  - Number of multiplications?

# Recursion: Implementation Details

- Methods in Java (or C, or Pascal, or …):
  - method-local variables stored in *stack* area
  - each *invocation* gets a *frame* on the *stack*
  - *frame* contains: parameters and locals
  - … and return value (eventually)
- The *call stack*:
  - frames *pushed* on top of stack, then *popped* off
  - “current” method computation is at top of stack

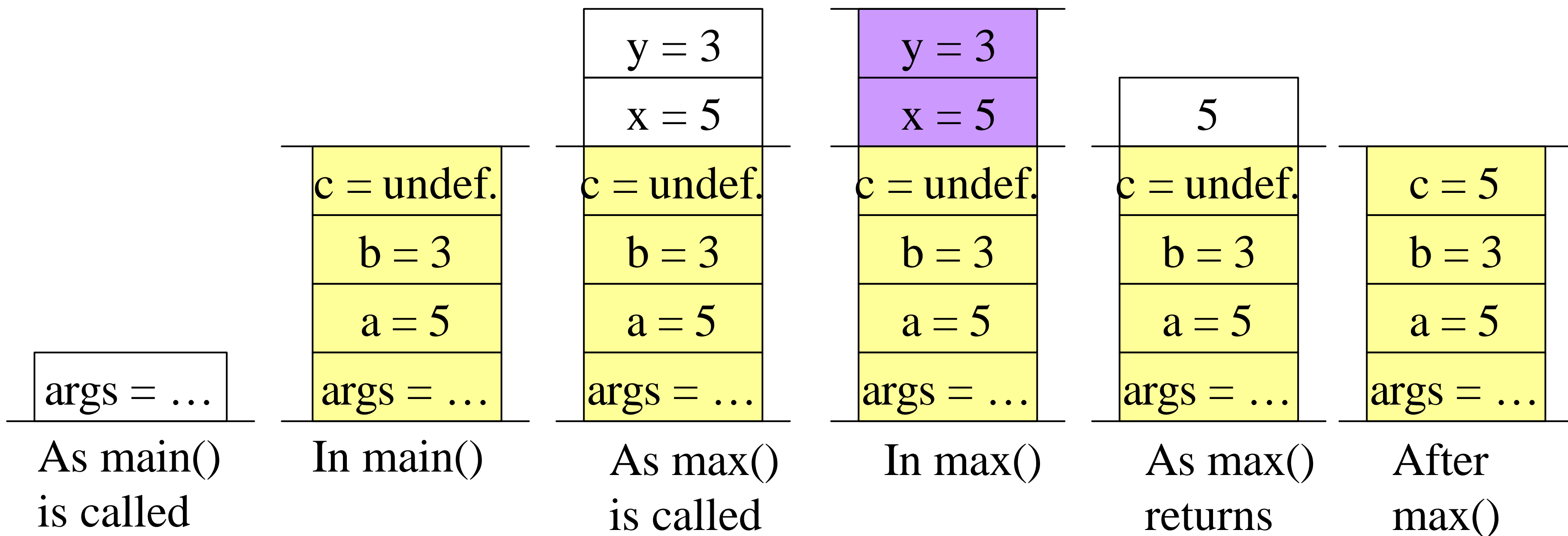
# Managing the Call Stack

- Non-recursive example:

```
static void main(String args[]) {
        int a = 5;
        int b = 3;
        int c = Math.max(a, b);
}
```

```
Static void main(String args[]) {
        int a = 5;
        int b = 3;
        int c = Math.max(a, b);
}
```

```
Static int max(int x, int y) {
        return x > y ? x : y;
}
```

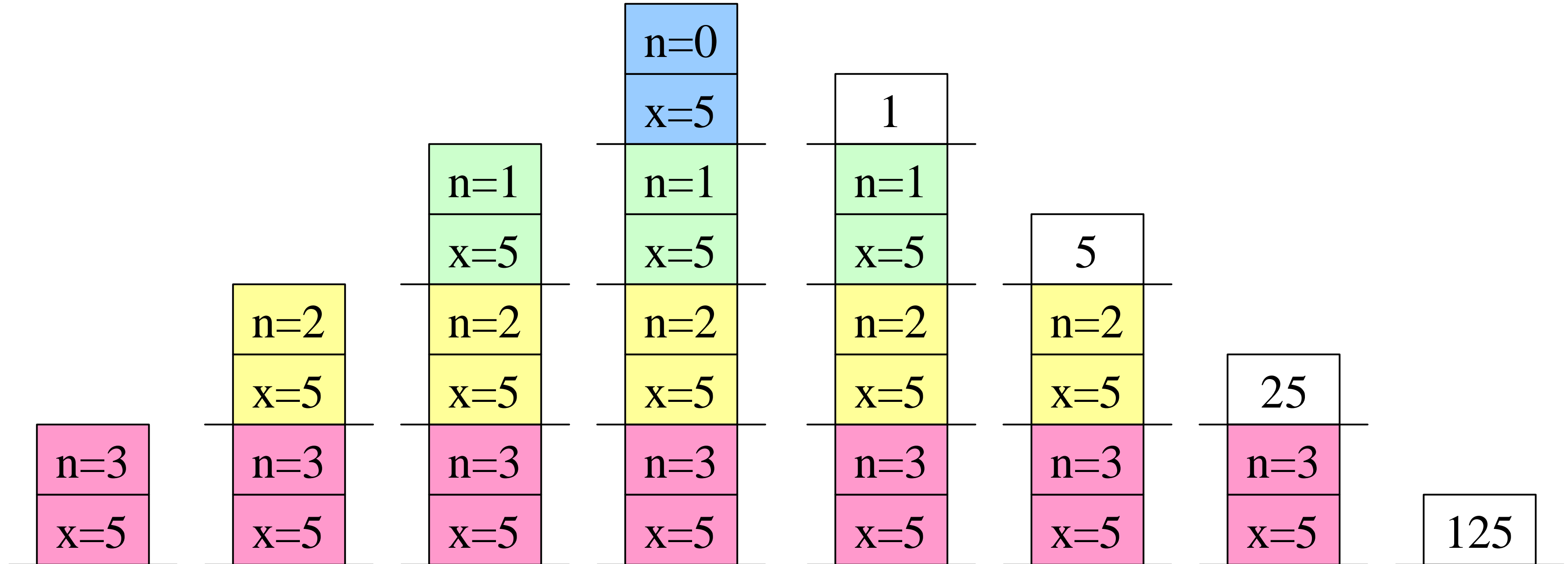


# Recursive Calls

- Call stack details for pow(2, 5)

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

```
static int pow(int x, int n) {
        if (n == 0) return 1;
        else return pow(x, n-1)*x;
}
pow(5, 3);
```



# Questions

- What location does *x* refer to in *pow()*?
  - Is it "statically" determined, or "dynamically"?
- How to keep track of which frame is "active"?
  - Always the top-most one
  - So keep a pointer to top, move up and down as needed. Call it the Stack Pointer (SP).
  - Also keep a pointer to "frame base", for debugging and efficiency. Call this the Frame Base Register (FBR)
  - All low-level machine-code relative to FBR or SP.
- How to know how big frame should be?
  - Static: compiler counts number of locals + params in the method
- Java is *type-safe*
  - Static: compiler tracks which slots have *ints*, *doubles, Strings, etc.*
  - e.g., it can catch all *type errors* at compile time