
Data Structures and Algorithms

Recursion

The mirrors

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy, and Dr. Low Kok Lim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Course website address is changed to <http://sakai.it.tdt.edu.vn>
- Slides “Practice Exercises” are eliminated.
- Course codes cs1010, cs1020, cs2010 are placed by 501042, 501043, 502043 respectively.

Objectives

1

- Strengthening the concept of recursion learned in 501042 (or equivalent)

2

- Demonstrating the application of recursion on some classic computer science problems

3

- Applying recursion on data structures

4

- Understanding recursion as a problem-solving technique known as divide-and-conquer paradigm

References



Book

- **Chapter 3:** Recursion: The Mirrors
- **Chapter 6:** Recursion as a Problem-Solving Technique, pages 337 to 345.



IT-TDT Sakai → 501043 website
→ Lessons

- <http://sakai.it.tdt.edu.vn>

Programs used in this lecture

- `CountDown.java`
- `ConvertBase.java`
- `SortedLinkedList.java`, `TestSortedList.java`
- `Combination.java`

Outline

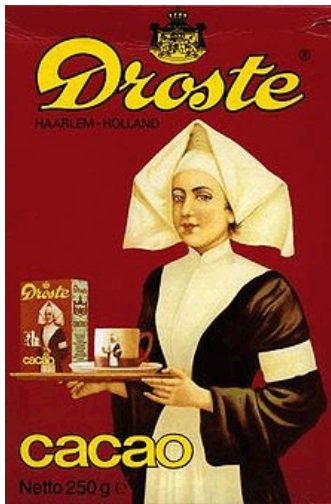
1. Basic Idea
2. How Recursion Works?
3. Examples
 - Count down
 - Display an integer in base b
 - Printing a Linked List
 - Printing a Linked List in reverse order
 - Inserting an element into a Sorted Linked List
 - Towers of Hanoi
 - Combinations: n choose k
 - Binary Search in a Sorted Array
 - Kth Smallest Number
 - Permutations of a string
 - The 8 Queens Problem

1 Basic Idea

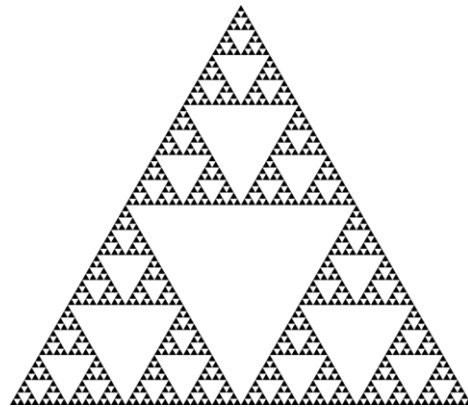
Also known as **a central idea in CS**

1.1 Pictorial examples

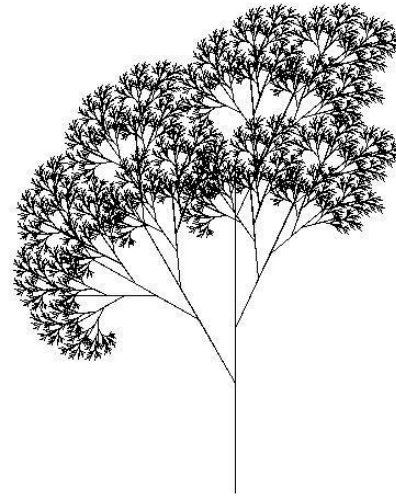
Some examples of recursion (inside and outside CS):



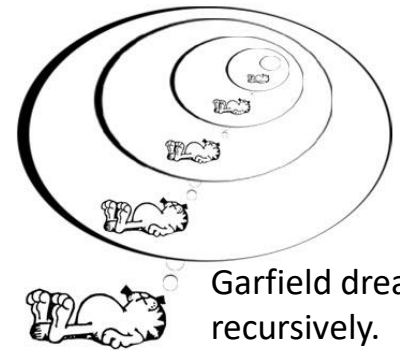
Droste effect



Sierpinski triangle



Recursive tree



Garfield dreaming recursively.

Recursion is the process of repeating items in a self-similar way but with smaller size.

1.2 Textual examples

Definitions based on recursion:

Recursive definitions:

1. A person is a **descendant** of another if
 - the former is the latter's child, or
 - the former is one of the **descendants** of the latter's child.
2. A **list of numbers** is
 - a number, or
 - a number followed by a **list of numbers**.

Dictionary entry:

Recursion: See recursion.

**To understand
recursion, you must
first understand
recursion.**

Recursive acronyms:

1. GNU = GNU's Not Unix
2. PHP = PHP: Hypertext Preprocessor



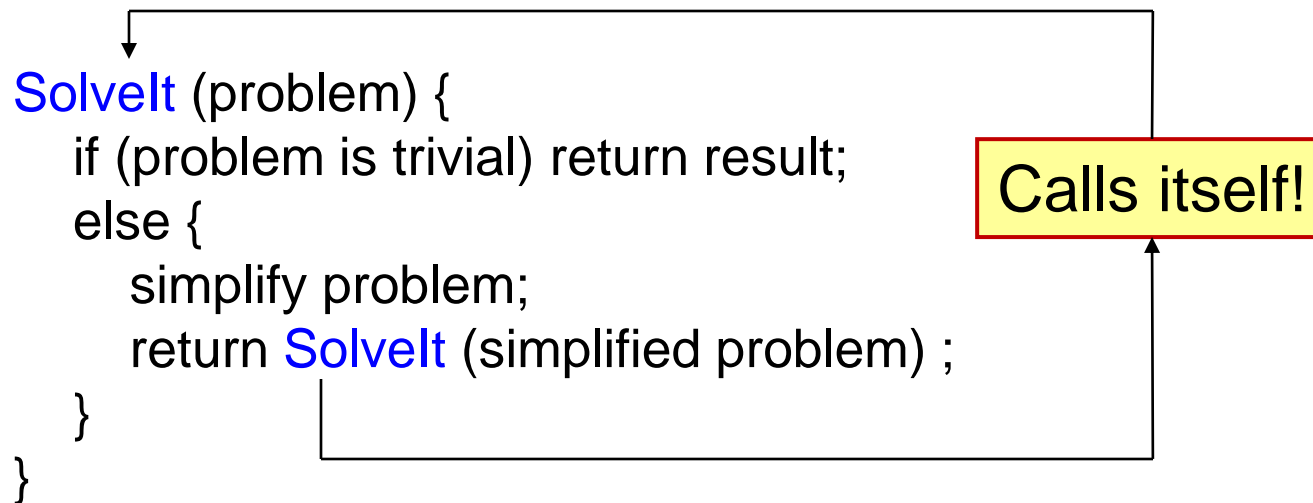
1.3 Divide-and-Conquer

- **Divide**: In top-down design (for program design or problem solving), break up a problem into **sub-problems of the same type**.
- **Conquer**: Solve the problem with the use of a function that **calls itself** to solve each sub-problem
 - one or more of these sub-problems are so **simple** that they can be solved directly without calling the function

**A paradigm where
the solution to a problem
depends on
solutions to smaller instances
of the SAME problem.**

1.4 Why recursion?

- Many algorithms can be expressed naturally in recursive form
- Problems that are complex or extremely difficult to solve using linear techniques may have simple recursive solutions
- It usually takes the following form:



2 How Recursion Works

Understanding Recursion

2.1 Recursion in 501042

- In 501042, you learned simple recursion
 - No recursion on data structures
 - Code consists of 'if' statement, no loop
 - How to trace recursive codes
- Examples covered in 501042
 - **Factorial** (classic example)
 - **Fibonacci** (classic example)
 - **Greatest Common Divisor** (classic example)
 - Other examples
 - Lecture slides and programs are available on 501043's "501042 Stuffs" page:
<http://sakai.it.tdt.edu.vn>

2.1 Recursion in 501042: Factorial (1/2)

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1) \times \dots \times 2 \times 1, & n > 0 \end{cases}$$

Iterative solution

```
// Precond: n >= 0
int fact(int n) {
    int result = 1;
    for (int i=1; i<=n; i++)
        result *= i;
    return result;
}
```

Remember to document pre-conditions,
which are common for recursive codes.

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1)!, & n > 0 \end{cases}$$

Recurrence relation

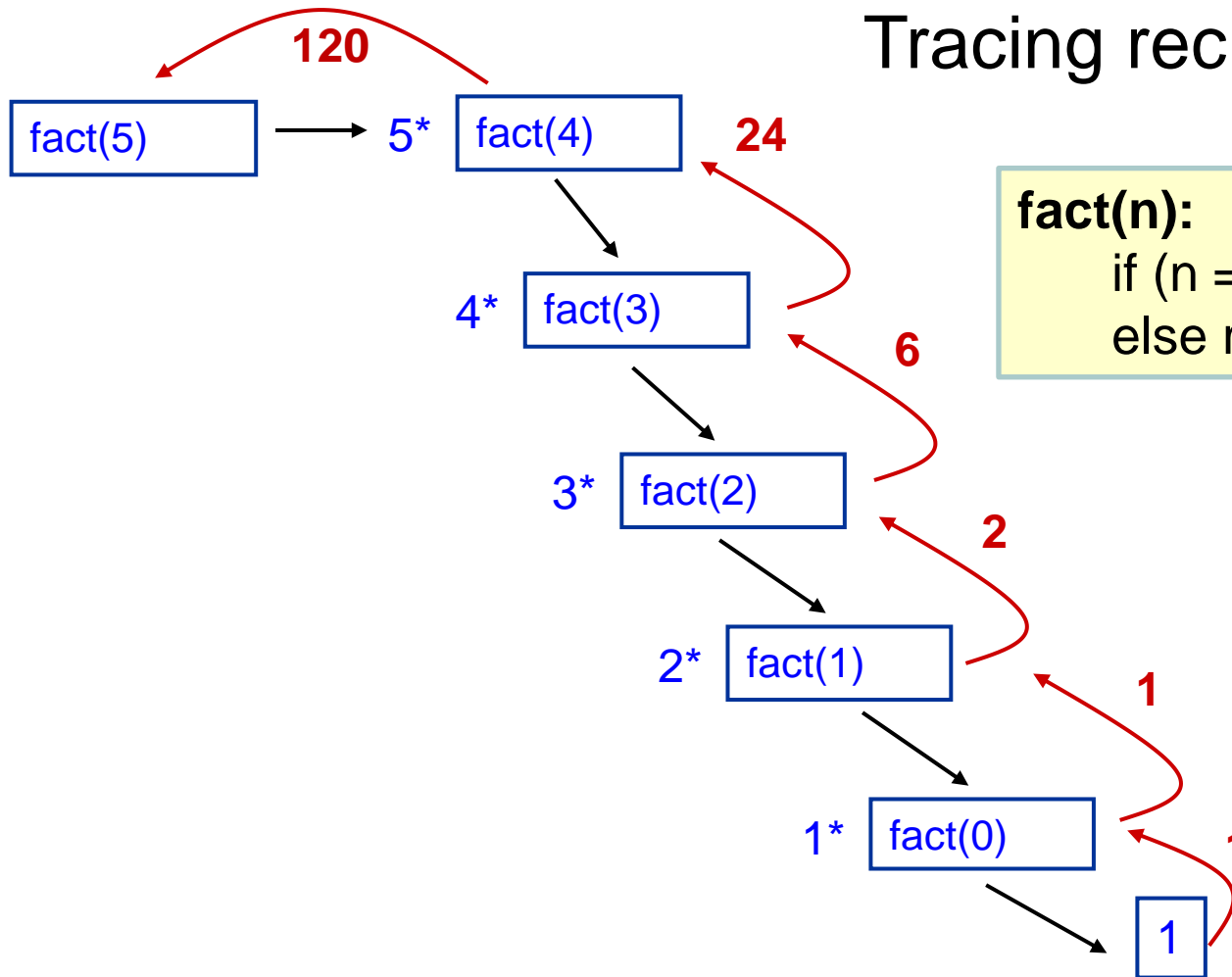
```
// Precond: n >= 0
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

Recursive
call

Base
case

2.1 Recursion in 501042: Factorial (2/2)

Tracing recursion



fact(n):
if (n == 0) return 1;
else return n * fact(n-1);

2.1 Recursion in 501042: Fibonacci (1/4)

- Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ...
 - The first two Fibonacci numbers are both 1 (arbitrary numbers)
 - The rest are obtained by adding the previous two together.
- Calculating the n^{th} Fibonacci number recursively:

$$\begin{aligned} \text{Fib}(n) &= 1 && \text{for } n=1, 2 \\ &= \text{Fib}(n-1) + \text{Fib}(n-2) && \text{for } n > 2 \end{aligned}$$

```
// Precond: n > 0
int fib(int n) {
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

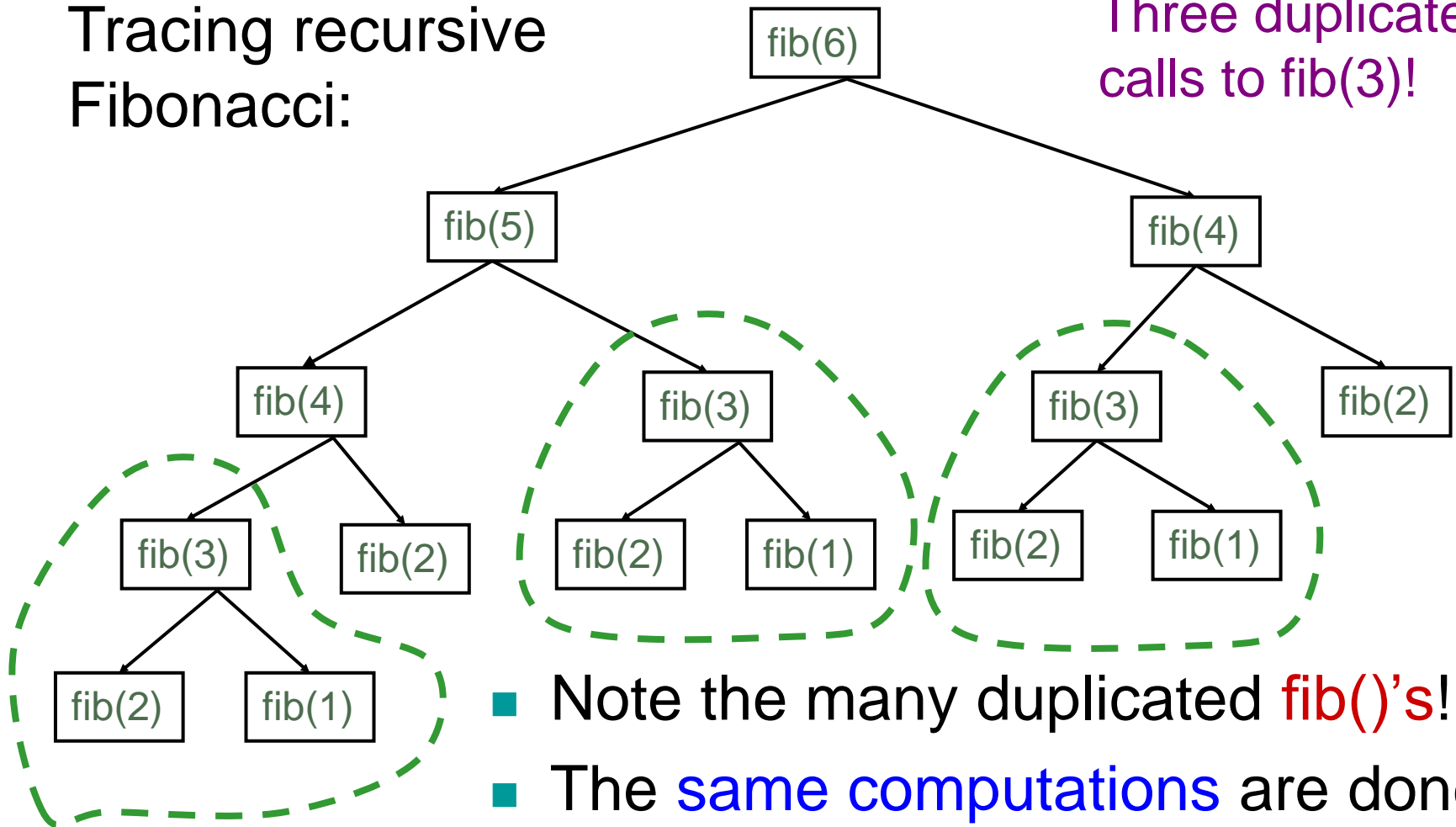
Elegant but extremely inefficient. Which is correct?

1. Recursion doesn't reach base case
2. A lot of repeated work
3. Should put recursive case above base case

2.1 Recursion in 501042: Fibonacci (2/4)

Tracing recursive
Fibonacci:

Three duplicate
calls to fib(3)!



- Note the many duplicated **fib()**'s!
- The **same computations** are done over and over again!

2.1 Recursion in 501042: Fibonacci (3/4)

Iterative Fibonacci

```
int fib(int n) {  
    if (n <= 2)  
        return 1;  
    else {  
        int prev1=1, prev2=1, curr;  
        for (int i=3; i<=n; i++) {  
            curr = prev1 + prev2;  
            prev2 = prev1;  
            prev1 = curr;  
        }  
        return curr;  
    }  
}
```

Q: Which part of the code is the key to the improved efficiency?

- (1) Part A (red)
- (2) Part B (blue)

2.1 Recursion in 501042: Fibonacci (4/4)

- Closed-form formula for Fibonacci numbers
- Take the ratio of 2 successive Fibonacci numbers (say A and B). The bigger the pair of numbers, the closer their ratio is to the **Golden ratio** φ which is $\approx 1.618034\dots$

A	2	3	5	8	...	144	233
B	3	5	8	13	...	233	377
B/A	1.5	1.666...	1.6	1.625	...	1.61805...	1.61802...

- Using φ to compute the Fibonacci number x_n :

$$x_n = \frac{\varphi^n - (1 - \varphi)^n}{\sqrt{5}}$$

See

<http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibFormula.html>

2.1 Recursion in 501042: GCD (1/2)

- Greatest Common Divisor of two integers a and b , where a and b are non-negative and not both zeroes
- Iterative method given in Practice Exercise 11

```
// Precond: a, b non-negative,  
//           not both zeroes  
int gcd(int a, int b) {  
    int rem;  
    while (b > 0) {  
        rem = a % b;  
        a = b;  
        b = rem;  
    }  
    return a;  
}
```

2.1 Recursion in 501042: GCD (2/2)

- Recurrence relation:

$$\text{gcd}(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \text{gcd}(b, a \% b), & \text{if } b > 0 \end{cases}$$

```
// Precond: a, b non-negative,  
//           not both zeroes  
int gcd(int a, int b) {  
    if (b == 0)  
        return a;  
    else  
        return gcd(b, a % b);  
}
```



2.2 Visualizing Recursion

Artwork credit: [ollie.olarte](https://www.ollieolarte.com/)

- It's easy to visualize the execution of non-recursive programs by stepping through the source code.
- However, this can be confusing for programs containing recursion.
 - Have to imagine **each call** of a method **generating a copy of the method (including all local variables)**, so that if the same method is called several times, several copies are present.

2.2 Stacks for recursion visualization

int j = fact(5)

fact(0)	1
fact(1)	1 × 1
fact(2)	2 × 1
fact(3)	3 × 2
fact(4)	4 × 6
fact(5)	5 × 24

Use

push() for new recursive call
pop() to return a value from
a call to the caller.


Example: fact (n)

```
if (n == 0) return 1;  
else return n * fact (n-1);
```

j = 120

2.3 Recipe for Recursion

Sometimes we call #1
the “**inductive step**”



To formulate a recursive solution:

1. **General (recursive) case**: Identify “**simpler**” instances of the same problem (so that we can make recursive calls to solve them)
2. **Base case**: Identify the “**simplest**” instance (so that we can solve it **without** recursion)
3. Be sure we are able to **reach** the “**simplest**” instance (so that we will not end up with **infinite recursion**)

2.4 Bad Recursion

```
funct(n) = 1                if (n==0)
          = funct(n - 2)/n   if (n>0)
```

Q: What principle does the above code violate?

1. Doesn't have a simpler step.
2. No base case.
3. Can't reach the base case.
4. All's good. It's a ~trick~!

3 Examples

How recursion can be used

3.1 Countdown

CountDown.java

```
public class CountDown {  
  
    public static void countDown(int n) {  
        if (n <= 0)    // don't use == (why?)  
            System.out.println ("BLAST OFF!!!!");  
        else {  
            System.out.println("Count down at time " + n);  
            countDown(n-1);  
        }  
    }  
  
    public static void main(String[] args) {  
        countDown(10);  
    }  
}
```

3.2 Display an integer in base b

- See [ConvertBase.java](#)

E.g. One hundred twenty three is 123 in base 10; 173 in base 8

```
public static void displayInBase(int n, int base) {  
    if (n > 0) {  
        displayInBase(n / base, base);  
        System.out.print(n % base);  
    }  
}
```

What is the precondition for parameter **base**?

Example 1:

$n = 123$, $base = 10$

$123/10 = 12$ $123 \% 10 = 3$

$12/10 = 1$ $12 \% 10 = 2$

$1/10 = 0$ $1 \% 10 = 1$

Answer: 123

Example 2:

$n = 123$, $base = 8$

$123/8 = 15$ $123 \% 8 = 3$

$15/8 = 1$ $15 \% 8 = 7$

$1/8 = 0$ $1 \% 8 = 1$

Answer: 173

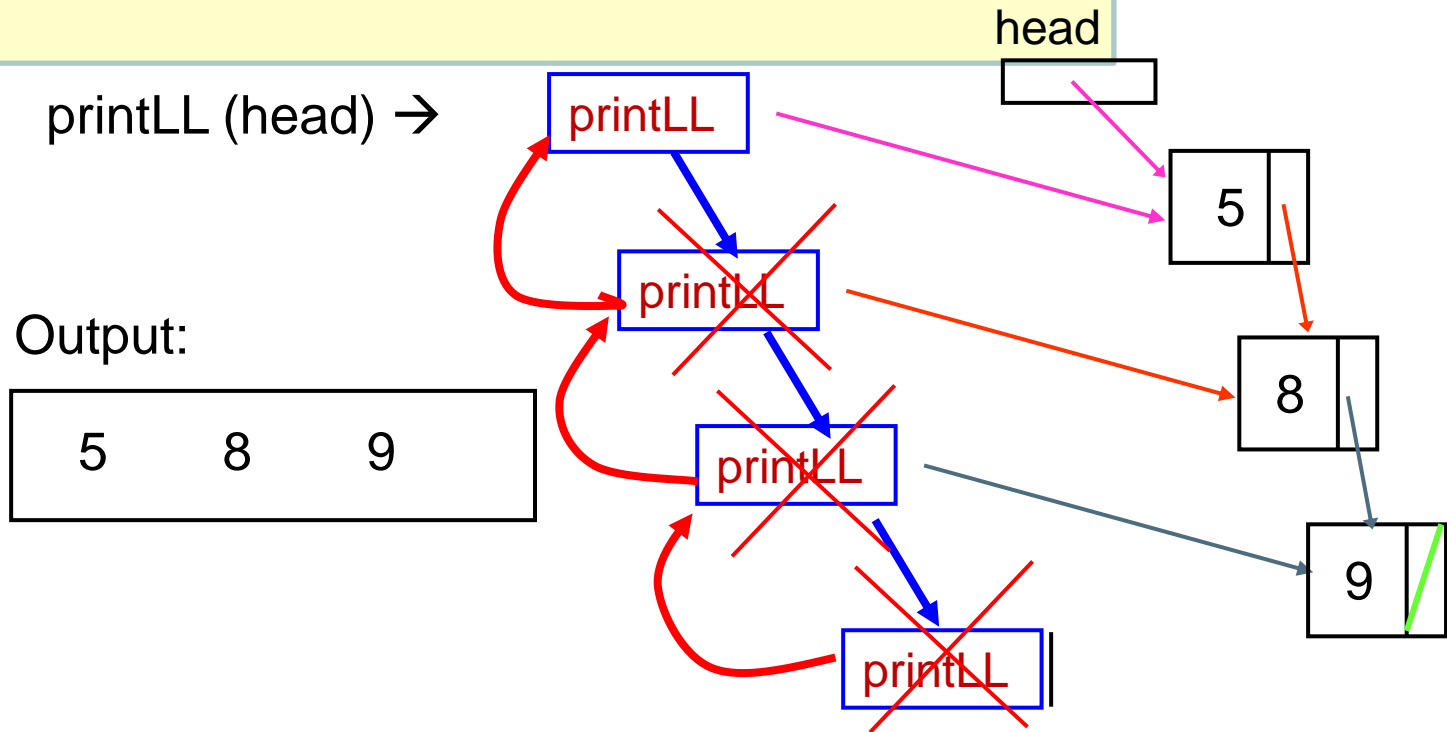
3.3 Printing a Linked List recursively

- See [SortedLinkedList.java](#) and [TestSortedList.java](#)

```
public static void printLL(ListNode n) {
    if (n != null) {
        System.out.print(n.value);
        printLL(n.next);
    }
}
```

Q: What is the base case?

Q: How about printing in reverse order?



3.4 Printing a Linked List recursively in reverse order

- See [SortedLinkedList.java](#) and [TestSortedList.java](#)

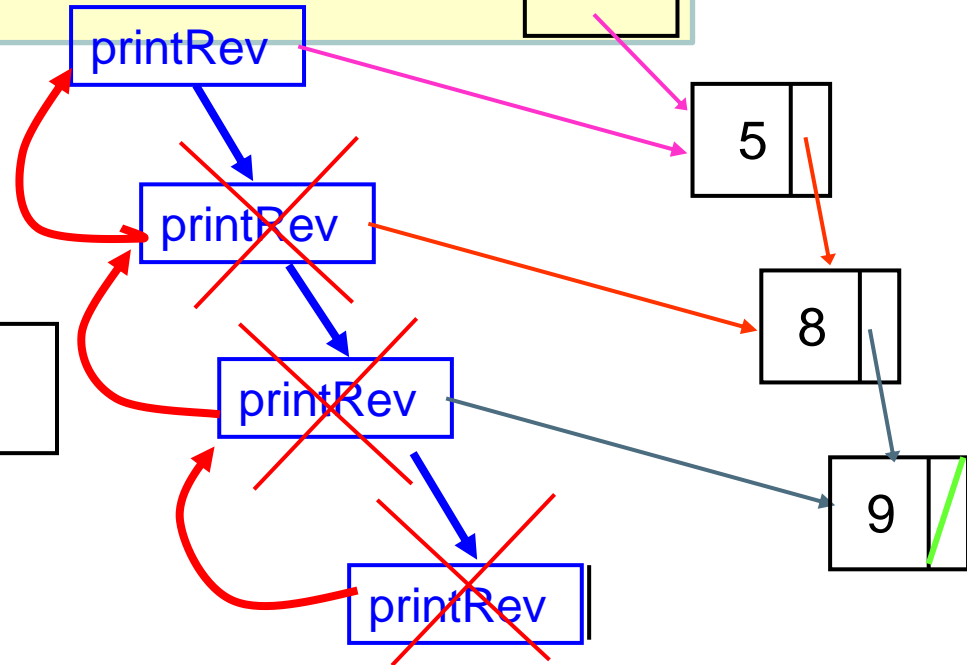
```
public static void printRev(ListNode n) {
    if (n != null) {
        printRev(n.next);
        System.out.print(n.value);
    }
}
```

Just change the name!
... Sure, right!

printRev(head) →

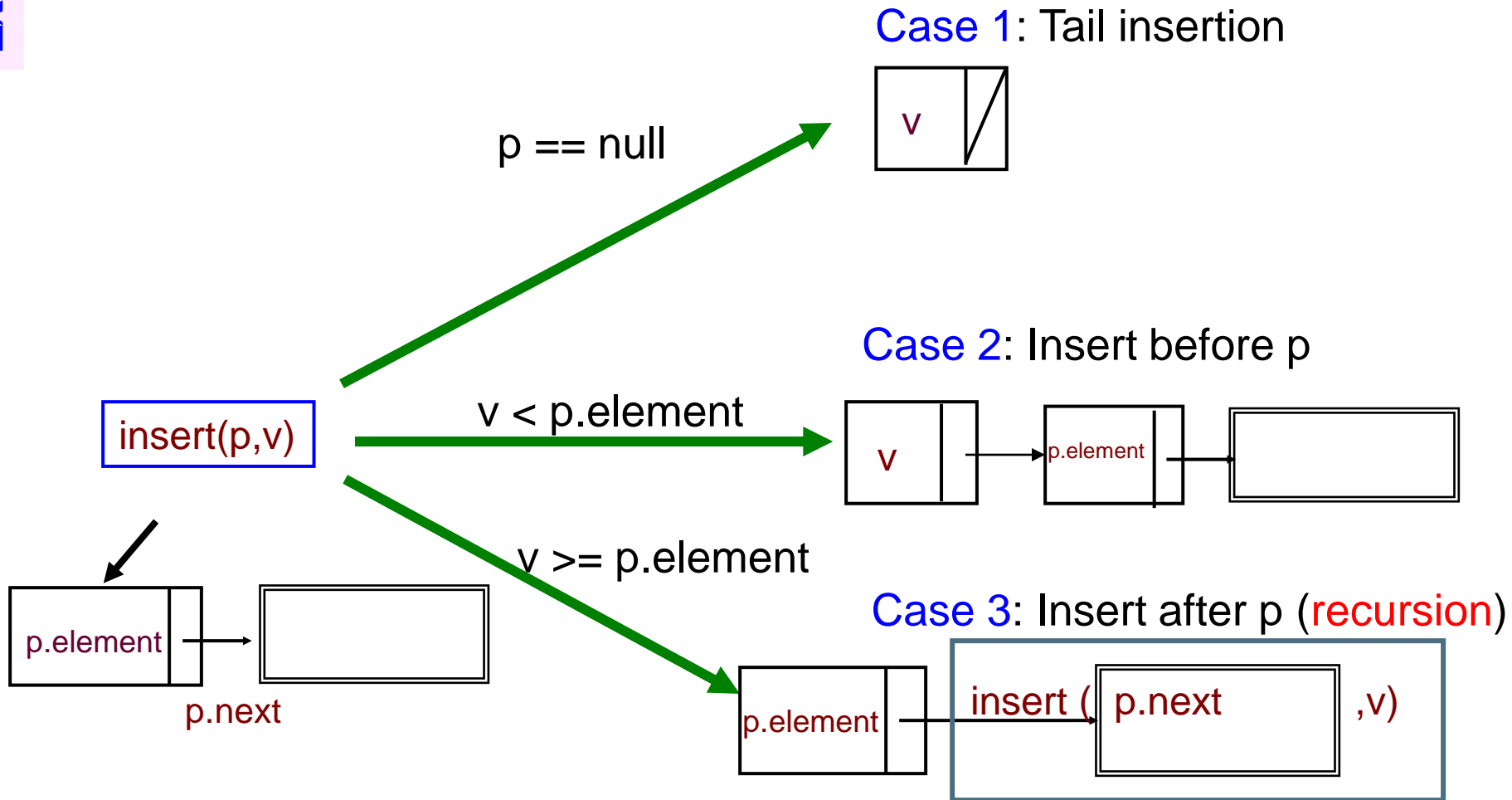
Output:

9	8	5
---	---	---



3.5 Sorted Linked List Insertion (1/2)

- Insert an item v into the sorted linked list with head p



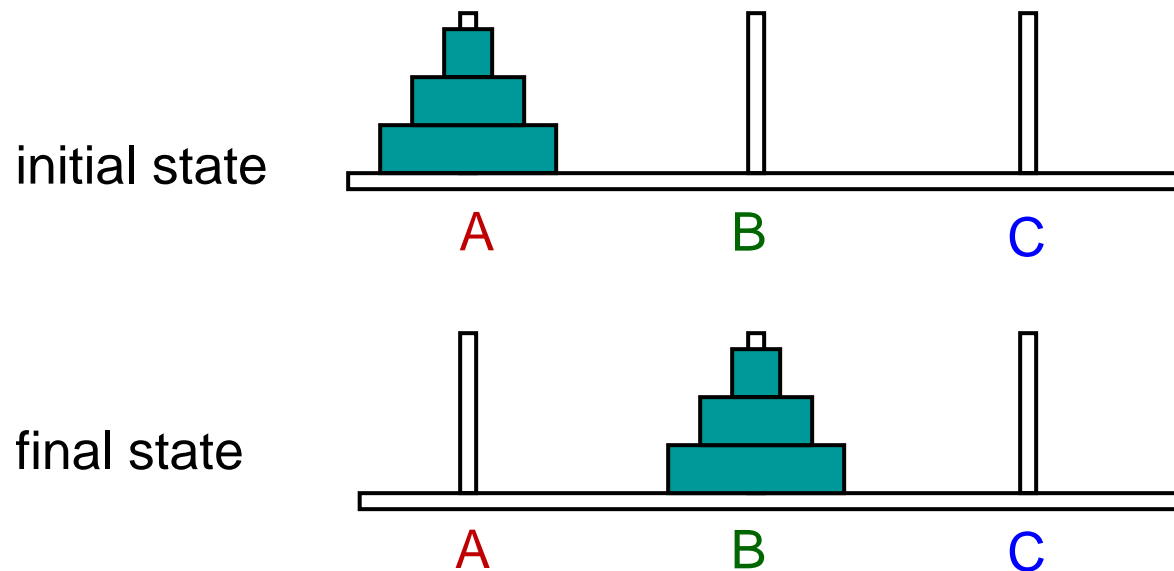
3.5 Sorted Linked List Insertion (2/2)

```
public static ListNode insert(ListNode p, int v) {  
    // Find the first node whose value is bigger than v  
    // and insert before it.  
    // p is the "head" of the current recursion.  
    // Returns the "head" after the current recursion.  
    if (p == null || v < p.element)  
        return new ListNode(v, p);  
    else {  
        p.next = insert(p.next, v);  
        return p;  
    }  
}
```

To call this method:
`head = insert(head, newItem);`

3.6 Towers of Hanoi

- Given a stack of discs on peg **A**, move them to peg **B**, one disc at a time, with the help of peg **C**.
- A larger disc cannot be stacked onto a smaller one.



Only the first 3 steps shown.

3.6 Towers of Hanoi – Quiz

- What's the **base case**?
 - ❑ A: 1 disc
 - ❑ B: 0 disc
- What's the **inductive step**?
 - ❑ A: Move the top $n-1$ disks to another peg
 - ❑ B: Move the bottom $n-1$ disks to another peg
- How many times do I need to call the inductive step?
 - ❑ A: Once
 - ❑ B: Twice
 - ❑ C: Three times



From en.wikipedia.org

3.6 Tower of Hanoi solution

```
public static void Towers(int numDisks, char src, char dest, char temp) {  
    if (numDisks == 1) {  
        System.out.println("Move top disk from pole " + src + " to pole " + dest);  
    } else {  
        Towers(numDisks - 1, src, temp, dest); // first recursive call  
        Towers(1, src, dest, temp);  
        Towers(numDisks - 1, temp, dest, src); // second recursive call  
    }  
}
```

3.6 Tower of Hanoi **iterative** solution (1/2)

```
public static void LinearTowers(int orig_numDisks, char orig_src,  
                                char orig_dest, char orig_temp) {  
    int numDisksStack[] = new int[100]; // Maintain the stacks manually!  
    char srcStack[] = new char[100];  
    char destStack[] = new char[100];  
    char tempStack[] = new char[100];  
    int stacktop = 0;  
    numDisksStack[0] = orig_numDisks; // Init the stack with the 1st call  
    srcStack[0] = orig_src;  
    destStack[0] = orig_dest;  
    tempStack[0] = orig_temp;  
    stacktop++;  
}
```

Complex!
This and the next slide are
only for your reference.

3.6 Tower of Hanoi **iterative** solution (2/2)

```
while (stacktop>0) {  
    stacktop--; // pop current off stack  
    int numDisks = numDisksStack[stacktop];  
    char src = srcStack[stacktop]; char dest = destStack[stacktop];  
    char temp = tempStack[stacktop];  
    if (numDisks == 1) {  
        System.out.println("Move top disk from pole "+src+" to pole "+dest);  
    } else {  
        /* Towers(numDisks-1,temp,dest,src); */ // second recursive call  
        numDisksStack[stacktop] = numDisks - 1;  
        srcStack[stacktop] = temp;  
        destStack[stacktop] = dest;  
        tempStack[stacktop++] = src;  
        /* Towers(1,src,dest,temp); */  
        numDisksStack[stacktop] = 1;  
        srcStack[stacktop] = src; destStack[stacktop] = dest;  
        tempStack[stacktop++] = temp;  
        /* Towers(numDisks-1,src,temp,dest); */ // first recursive call  
        numDisksStack[stacktop] = numDisks - 1;  
        srcStack[stacktop] = src; destStack[stacktop] = temp;  
        tempStack[stacktop++] = dest;  
    }  
}
```

Q: Which version runs faster?

A: Recursive

B: Iterative (this version)

3.6 Towers of Hanoi

Towers(4, src, dest, temp)



numDiskStack



srcStack



destStack



tempStack

3.6 Time Efficiency of Towers()

Num of discs, n	Num of moves, f(n)	Time (1 sec per move)
1	1	1 sec
2	3	3 sec
3	$3+1+3 = 7$	7 sec
4	$7+1+7 = 15$	15 sec
5	$15+1+15 = 31$	31 sec
6	$31+1+31 = 63$	1 min
...
16	65,536	18 hours
32	4.295 billion	136 years
64	1.8×10^{10} billion	584 billion years
N	$2^N - 1$	

3.7 Being choosy...



“Photo” credits: [Torley](#)
(this pic is from 2nd life)

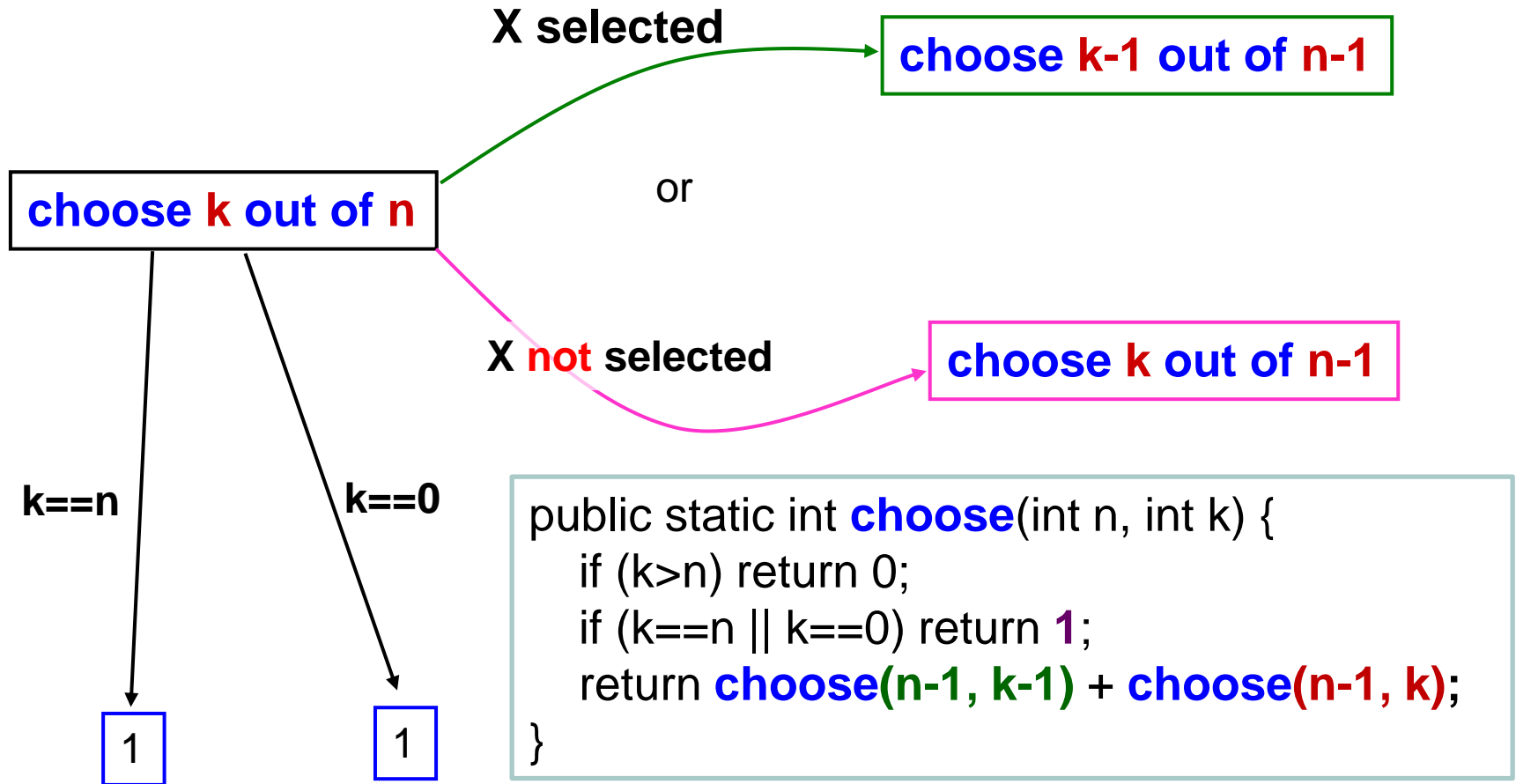
Suppose you visit an ice cream store with your parents.

You’ve been good so they let you choose **2** flavors of ice cream.

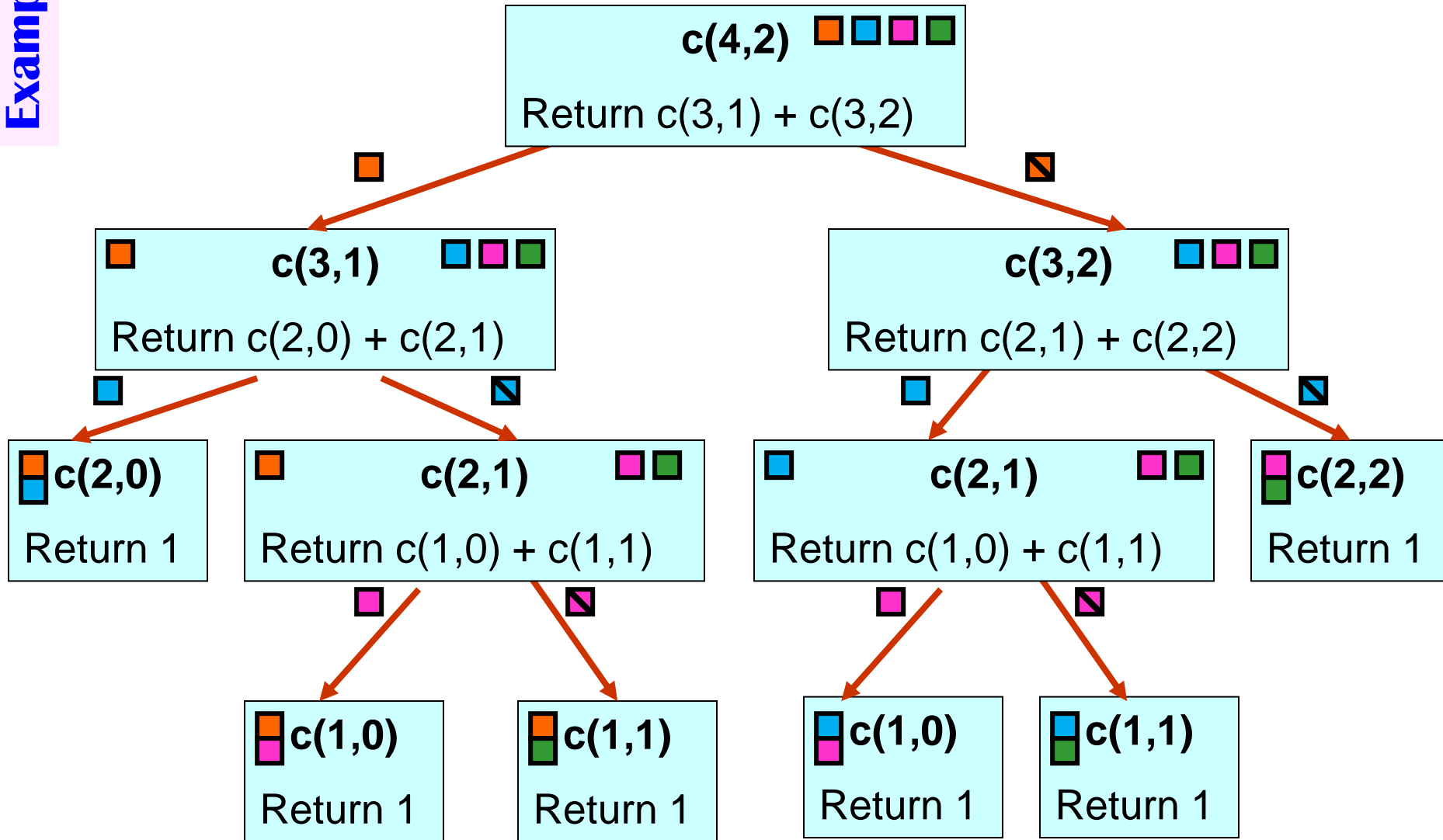
The ice cream store stocks **10** flavors today. **How many different ways** can you choose your ice creams?

3.7 n choose k

- See [Combination.java](#)



3.7 Compute $c(4,2)$



3.8 Searching within a sorted array

- **Idea:** narrow the search space by **half** at every iteration until a single element is reached.

Problem: Given a **sorted** int array **a** of n elements and int **x**, determine if **x** is in **a**.

a =

1	5	6	13	14	19	21	24	32
---	---	---	----	----	----	----	----	----

x = 15

3.8 Binary Search by Recursion

```
public static int binarySearch(int [] a, int x, int low, int high)
                                throws ItemNotFound {
    // low: index of the low value in the subarray
    // high: index of the highest value in the subarray
    if (low > high) // Base case 1: item not found
        throw new ItemNotFound("Not Found");

    int mid = (low + high) / 2;

    if (x > a[mid])
        return binarySearch(a, x, mid + 1, high);
    else if (x < a[mid])
        return binarySearch(a, x, low, mid - 1);
    else
        return mid; // Base case 2: item found
}
```

Q: Here, do we assume that the array is sorted in ascending or descending order?

A: Ascending

B: Descending

3.8 Auxiliary functions for recursion

- Hard to use this function as it is.
- Users just want to find something in an array. They don't want to (or may not know how to) specify the **low** and **high** indices.
 - Write an auxiliary function to call the recursive function
 - Using **overloading**, the auxiliary function can have the same name as the actual recursive function it calls

Auxiliary
function

```
boolean binarySearch(int[] a, int x) {  
    return binarySearch(a, x, 0, a.length-1);  
}
```

Recursive function

3.9 Find k^{th} smallest (unsorted array)

```
public static int kthSmallest(int k, int[] a) { // k >= 1
```

```
    // Choose a pivot element p from a[]
```

```
    // and partition (how?) the array into 2 parts where
```

```
    // left = elements that are <= p
```

```
    // right = elements that are > p
```

```
    ...
```

```
    int numLeft = sizeOf(left);
```

```
    if (1_____) 3_____;
```

```
    if (2_____) {
```

```
        return 4_____;
```

```
    else
```

```
        return 5_____;
```

```
}
```

Map the lines to the slots

A: 1i, 2ii, 3iii, 4iv, 5v

B: 1i, 2ii, 3v, 4iii, 5iv

C: 1ii, 2i, 3v, 4iii, 5iv

D: 1i, 2ii, 3v, 4iv, 5iii

where

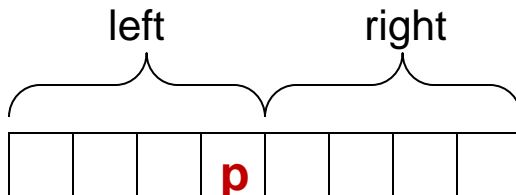
i. $k == \text{numLeft}$

ii. $k < \text{numLeft}$

iii. return **kthSmallest**(k, left);

iv. return **kthSmallest**(k - numLeft, right);

v. return p;



3.10 Find all Permutations of a String (1/3)

- For example, if the user types a word say *east*, the program should print all **24** permutations (anagrams), including *eats*, *etas*, *teas*, and non-words like *tsae*.
- Idea to generate all permutation:
 - Given *east*, we would place the **first** character i.e. **e** in front of all **6** permutations of the other **3** characters *ast* — *ast*, *ats*, *sat*, *sta*, *tas*, and *tas* — to arrive at *east*, *eats*, *esat*, *esta*, *etas*, and *etsa*, then
 - we would place the second character, i.e. **a** in front of all 6 permutations of *est*, then
 - the **third** character i.e. **s** in front of all 6 permutations of *eat*, and
 - finally the **last** character i.e. **t** in front of all 6 permutations of *eas*.
 - Thus, there will be **4** (the size of the word) **recursive calls** to display all permutations of a four-letter word.
- Of course, when we're going through the permutations of the **3**-character string e.g. *ast*, we would follow the same procedure.

3.10 Find all Permutations of a String (2/3)

- Recall overloaded `substring()` methods in `String` class

String	substring (int <code>beginIndex</code>) Returns a new string that is a substring of this string. The substring begins with the character at <code>beginIndex</code> and extends to the end of this string.
String	substring (int <code>beginIndex</code> , int <code>endIndex</code>) Returns a new string that is a substring of this string. The substring begins at <code>beginIndex</code> and extends to the character at index <code>endIndex - 1</code> . Thus the length of the substring is <code>endIndex - beginIndex</code> .

3.10 Find all Permutations of a String (3/3)

```

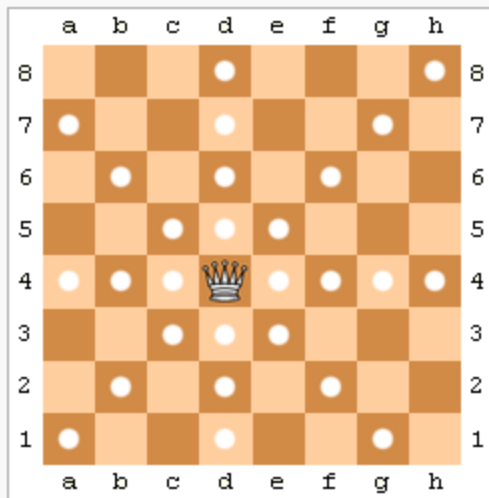
public class Permutations {
    public static void main(String args[]) {
        permuteString("", "String");
    }

    public static void permuteString(String beginningString, String endingString) {
        if (endingString.length() <= 1)
            System.out.println(beginningString + endingString);
        else
            for (int i = 0; i < endingString.length(); i++) {
                try {
                    String newString = endingString.substring(0,i) + endingString.substring(i+1);
                    // newString is the endingString but without character at index i
                    permuteString(beginningString + endingString.charAt(i), newString);
                } catch (StringIndexOutOfBoundsException exception) {
                    exception.printStackTrace();
                }
            }
    }
}

```

Exercise: Eight Queens Problem

- Place **eight Queens** on the chess board so that they cannot attack one another



Possible moves of the queen are shown

- **Q:** How do you formulate this as a recursion problem?
Work with a partner on this.

http://en.wikipedia.org/wiki/Eight_queens_puzzle

Backtracking

- Recursion and stacks illustrate a key concept in search: **backtracking**
- We can show that the recursion technique can exhaustively search all possible results in a systematic manner
- Learn more about searching spaces in other CS classes.

More Recursion later

- You will see more examples of recursion later when we cover more advanced sorting algorithms
 - Examples: Quick Sort, Merge Sort

5 Summary

- **Recursion** – The Mirrors
- **Base Case:**
 - Simplest possible version of the problem which can be solved easily
- **Inductive Step:**
 - Must simplify
 - Must arrive at some base case
- Easily visualized by a Stack
- Operations **before** and **after** the recursive calls come in **FIFO** and **LIFO** order, respectively
- Elegant, but **not** always the best (most efficient) way to solve a problem

End of file