CS1020 Lecture Note #5: **Recursion**

The Mirrors

Lecture Note #5: Recursion

- **Objectives:**
  - To explain how recursion work
  - To demonstrate the application of recursion on some classic computer science problems
  - To understand recursion as a problem solving technique, used in divide-and-conquer paradigm

Outline

- Recursion - Basic idea
  - Iteration versus Recursion
- How Recursion Works?
  - Visualizing the execution of a recursive program
- Examples
  - Printing a Linked-List
  - Inserting an Element into a Sorted Linked-List
  - Towers of Hanoi
  - Combinations
  - Binary Search in a Sorted Array
  - Kth Smallest Number
  - Fibonacci Numbers
  - Permutations
  - The 8 Queens Problem

1 Basic Idea

Also known as a central idea in CS

1.1 Concept

- Divide: In top-down design, 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 method where the solution to a problem depends on solutions to <u>smaller instances</u> of the <u>SAME</u> problem.

1.2 Why recursion?

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

```
Solve_It (problem) {
    if (problem is trivial) return result;
    else {
        simplify problem;
        return Solve_It (simplified problem) ;
    }
}
```

Calls itself!

Example 1

1.3 Countdown

CountDown.java

```
import java.util.*;

class CountDown {
    public static void count_down(int n) {
        if (n <= 0)                               // don’t use ==, why?
            System.out.println (“BLAST OFF!!!!”);
        else {
            System.out.println( “Count down at time ”+ n);
            count_down(n-1);
        }
    }

    public static void main(String[] args) {
        count_down(10);
    }
}
```

Example 2

1.4 Greatest Common Divisor (GCD)

```
public static int gcd(int n1, int n2) {
   // Assume n1>0, n2>=0, and n1>=n2

   n1 = Math.abs(n1);      // this is not
   n2 = Math.abs(n2);      // very good
   if (n1 < n2)
      return gcd(n2, n1);
   if (n2 == 0)
      return n1;
   return gcd(n2, n1 % n2);
}
```

Example 3

1.5 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);    // integer division
      System.out.print (n % base);      // remainder
   }
}
```

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

1.6 Factorial

- fact(n), the product of numbers from 1 to n, is defined as:

  fact(n) = n * (n-1) * (n-2) * ... * 2 * 1, and

  fact(0) = 1

- Using recursion, it can be defined as

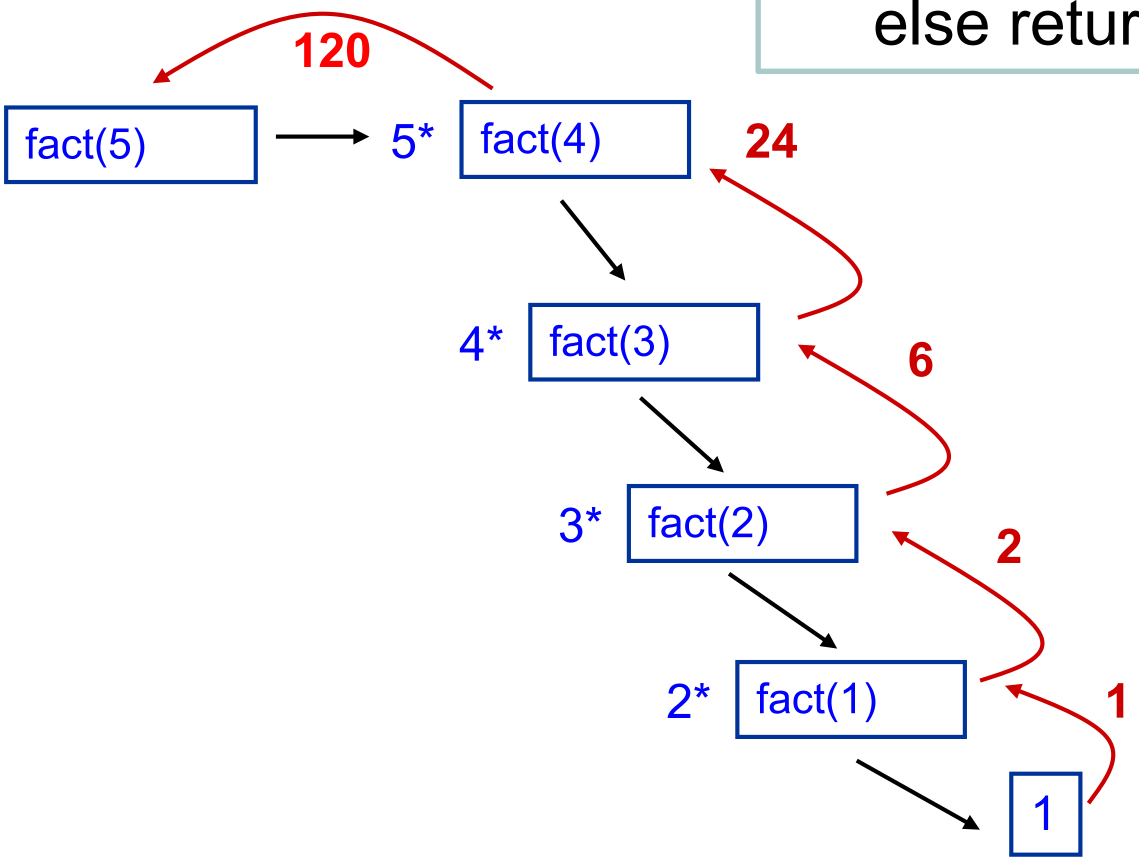
```
fact(n) = 1               if (n==0) // simple sub-problem
        = n*fact(n-1)     if (n>0)  // calls itself
```

2 How Recursion Works

Understanding Recursion

2.1 Tracing factorial

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



2.2 Visualizing Recursion

Artwork credit: ollie.olarte

- 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 function generating a copy of the function (including all local variables), so that if the same function is called several times, several copies are present.

Quiz Time

Q: We've already learned an ADT that makes recursion easy to visualize. What is it?

- A: Stacks
- B: Queues
- C: Deques (**double-ended queues**)
- D: Both Stacks and Queues, so my answer is Lists

2.3 Stacks for recursion visualization

int j = fact(5)

fact(1)	1
fact(2)	2 x1
fact(3)	3 x2
fact(4)	4 x6
fact(5)	5 x24

j =120

Use

push() for new recursive call

pop() to return a value from a call to the caller.

Example: fact (n):

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

2.4 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)

2.5 Not a Good Recursion

$$\text{funct}(n) = \begin{cases} 1 & \text{if } (n==0) \\ \text{funct}(n-2)/n & \text{if } (n>0) \end{cases}$$

Q: What principle does the above principle 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

Example 4

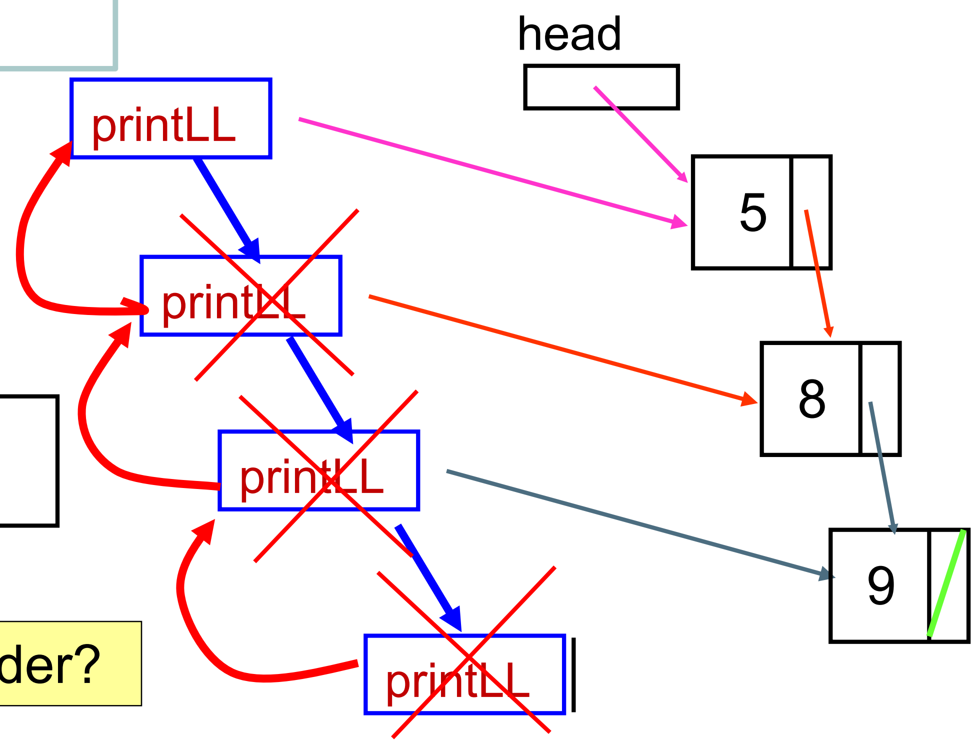
Printing a Linked List recursively

- 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?

printLL (head) →



Output:

5 8 9

Q: How about printing in reverse order?

Example 5

Printing a Linked List in reverse order

- See SortedLinkedList.java and TestSortedList.java

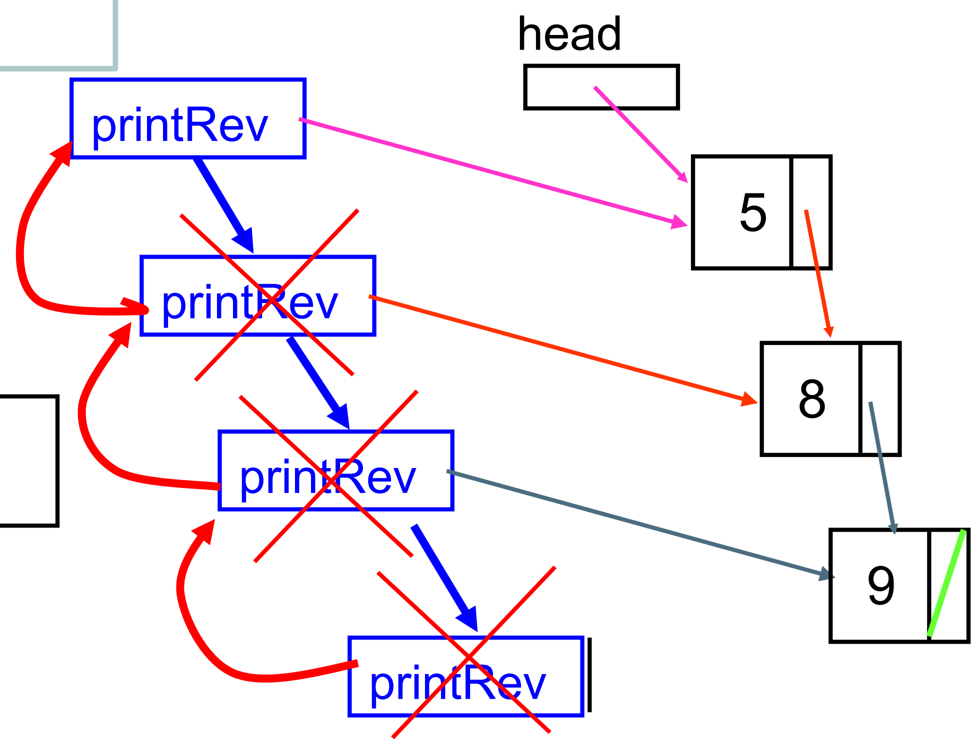
```
public static void printRev (ListNode n) {
  if (n!=null) {
    printRev (n.next);
    System.out.print(n.value)
  }
}
```

Just change the name!
… Sure, right!

printRev(head) →

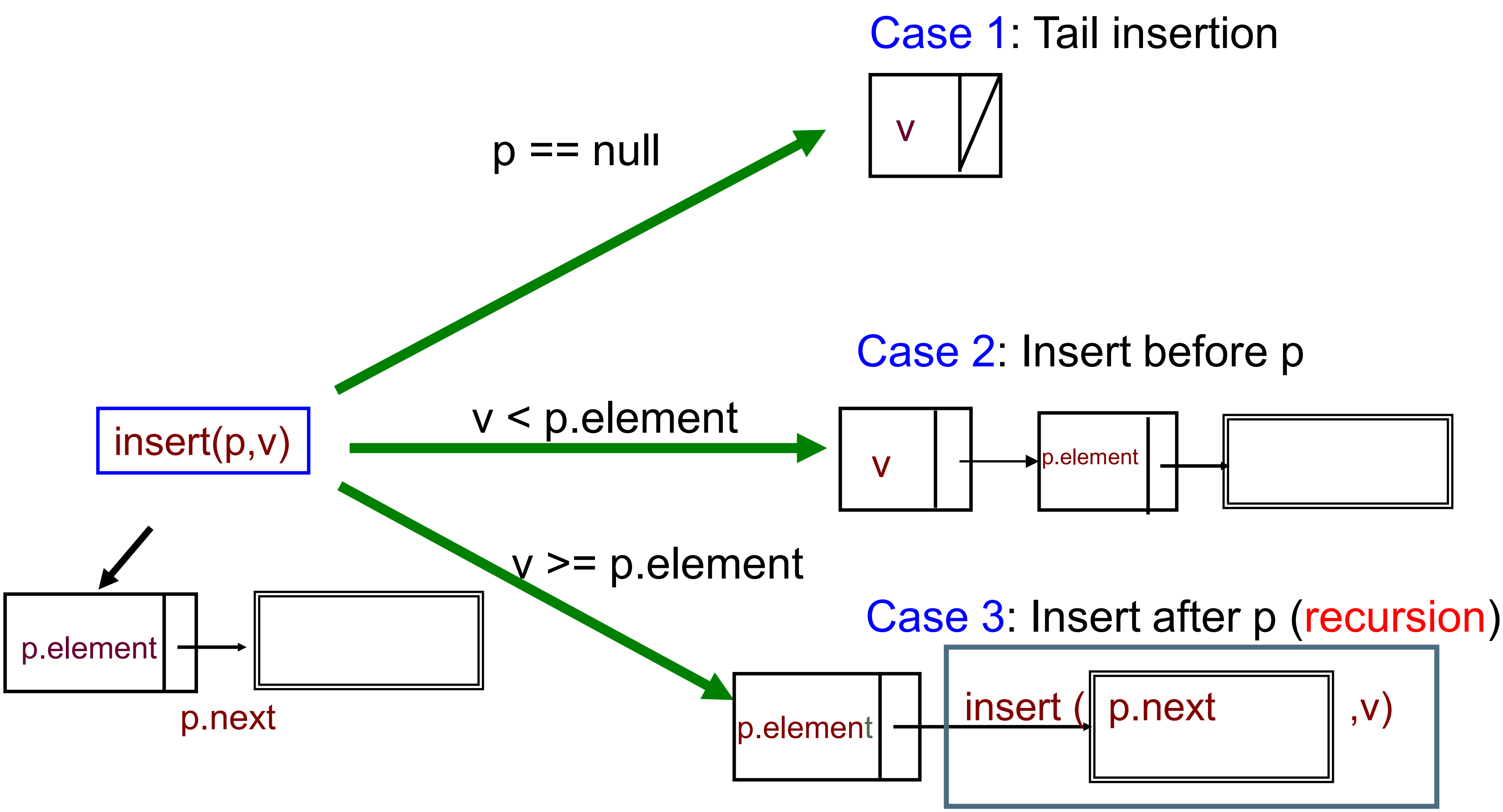
Output:

9 8 5



Example 6

Sorted Linked List Insertion



Example 6

Recursive Insertion

```
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;
   }
}
```

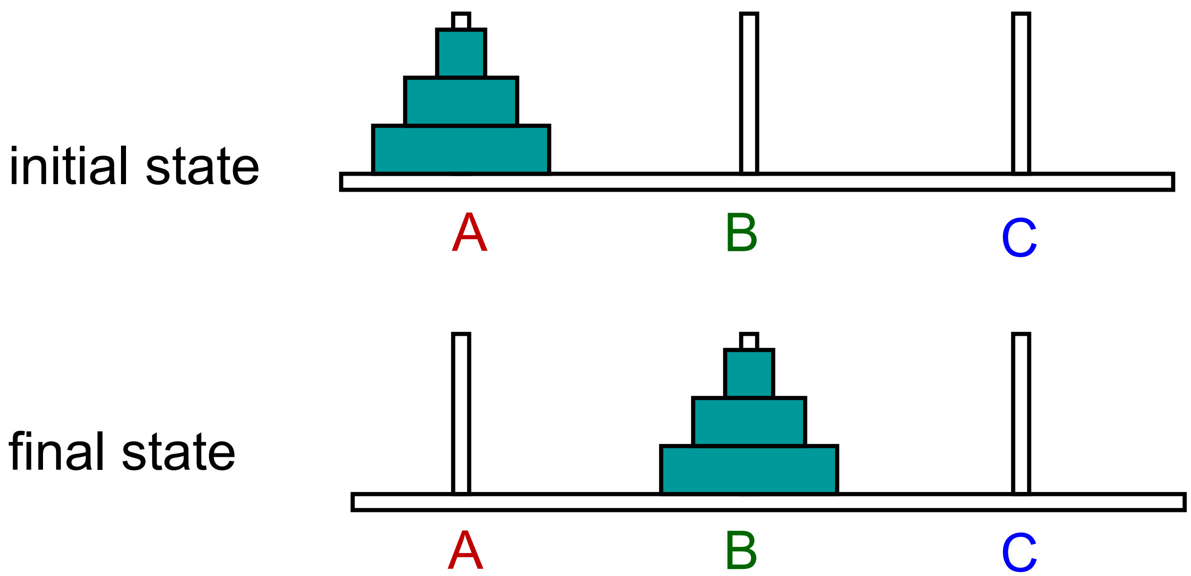
To call this method:

```
head = insert(head, newItem);
```

Example 7

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.

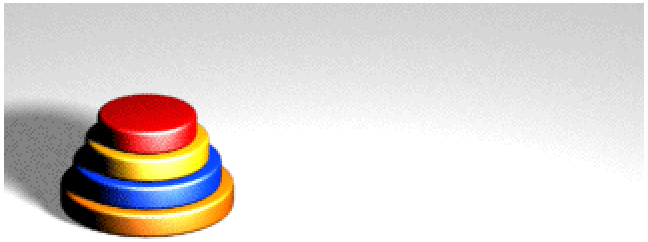


Only the first 3 steps shown.

Example 7

Quiz Time – Towers of Hanoi

- What's the base case?
  - A: 1 disc
  - B: 0 discs
- 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

Example 7

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
  }
}
```

Example 7

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++;
```

Example 7

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)

Example 7

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**

Example 8

Being choosy…

"Photo" credits: Torley
(this pic is from 2nd life)

Suppose you visit an ice cream store with your parents.

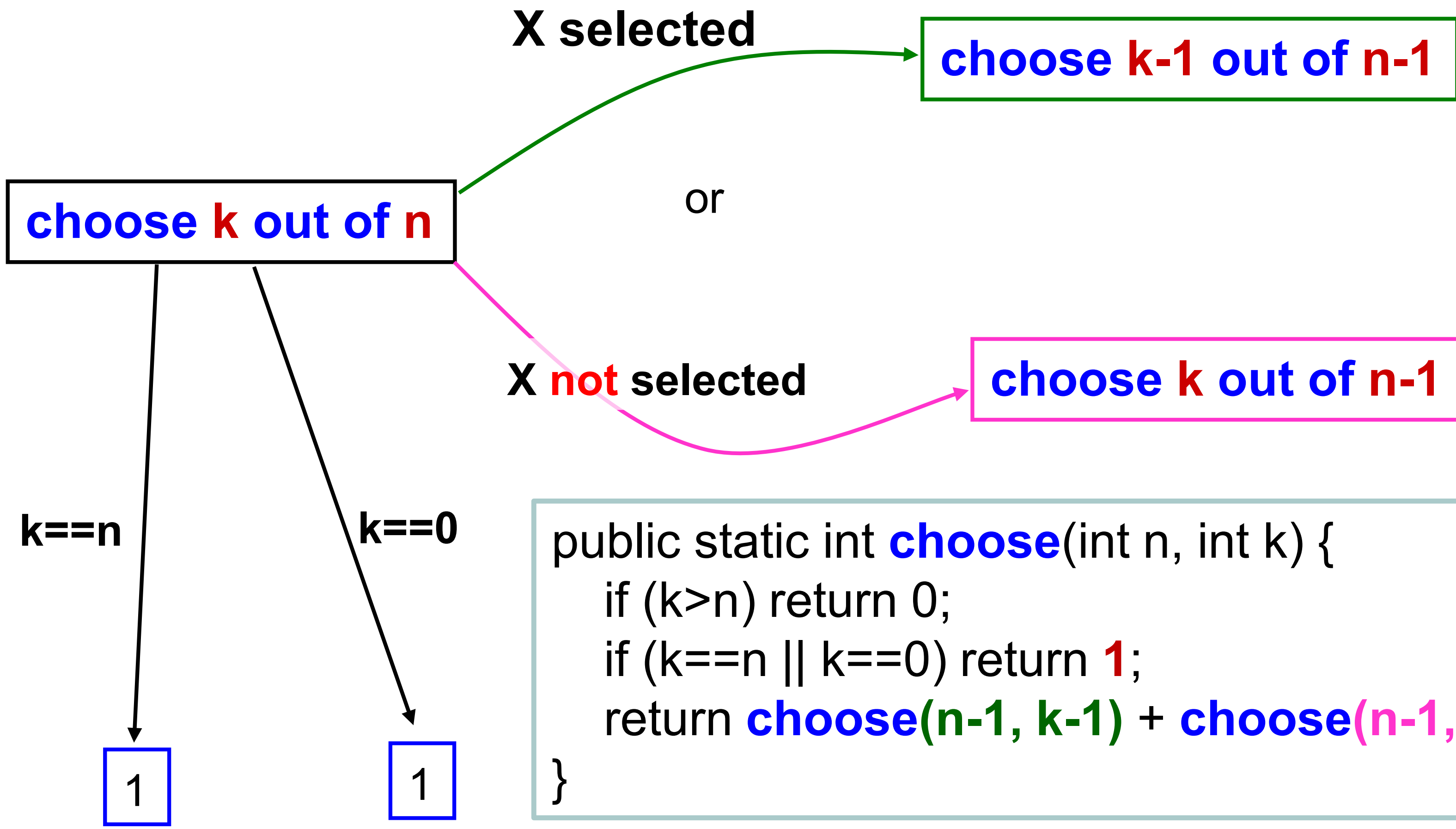
You've been good so they let you choose 2 flavors of ice cream.

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

Example 8

n choose k

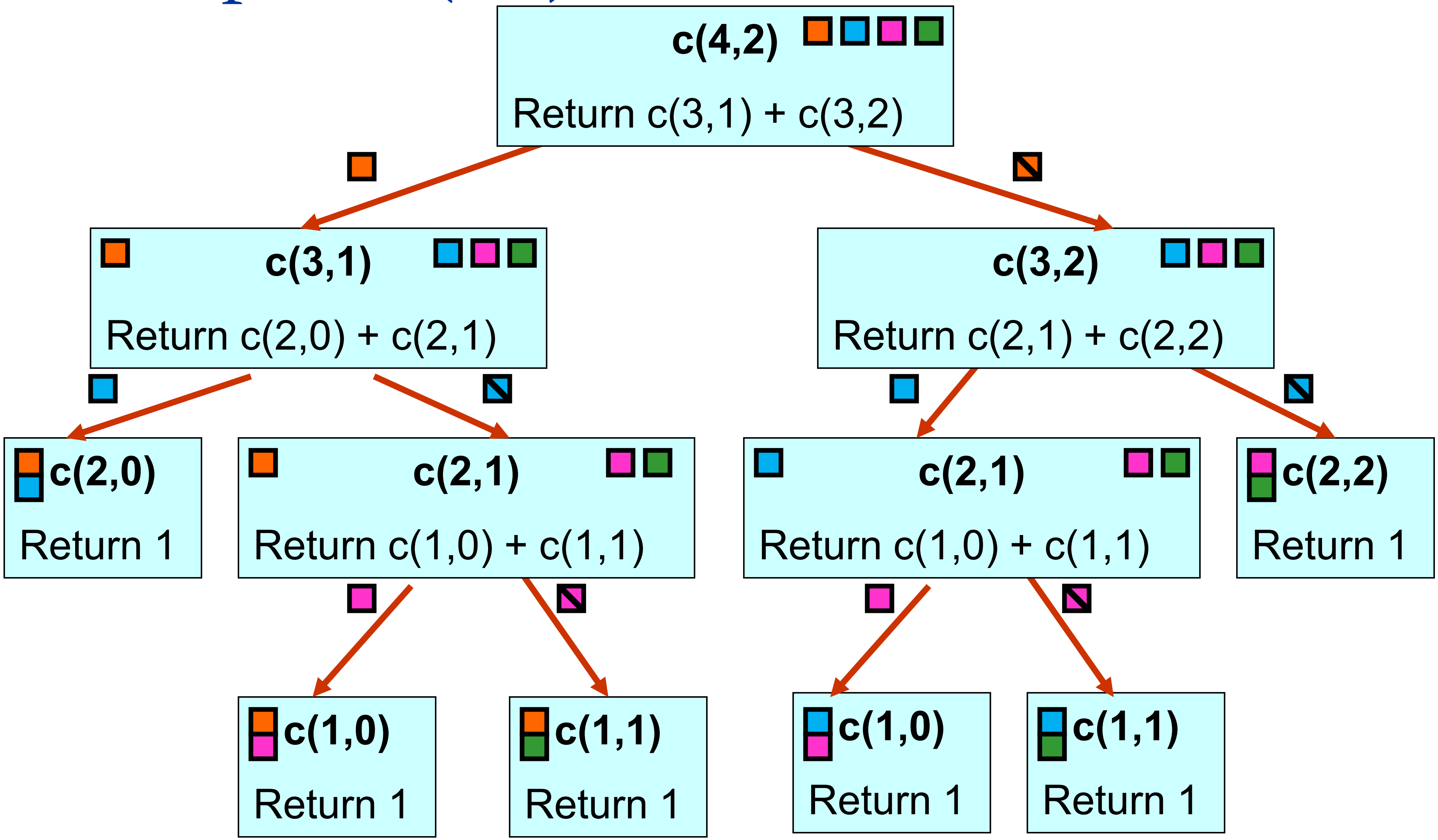
- See Combination.java




```java
public static int choose(int n, int k) {
    if (k>n) return 0;
    if (k==n || k==0) return 1;
    return choose(n-1, k-1) + choose(n-1, k);
}
```

Example 8

Compute c(4,2)



The final answer is the sum of the base cases.

Example 9

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

Example 9

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: Do we assume that the array is sorted in ascending or in descending order?

A: Ascending

B: Descending

Example 9

Starting 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.
  - Use overloading!

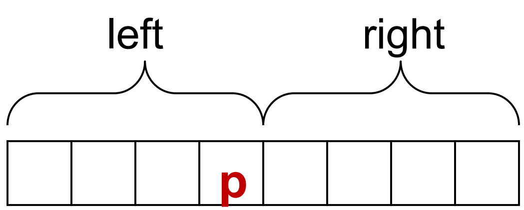
```
boolean binarySearch(int[] a, int x) {
  return binarySearch(a, x, 0, a.length-1);
}
```

Example 10

Find the kth smallest number (unsorted array a)

```
public static int ksmall(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 smaller than or equal to p
    //     right = elements that are larger than p
     …
    int numLeft = sizeOf(left);

    if (1____________) 3_____________;
    if (2____________) {
        return 4____________;
    else
        return 5____________;
}
```



Quiz Time! 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 == numLeft
ii. k < numLeft
iii. return ksmall(k, left);
iv. return ksmall(k – numLeft, right);
v. return p;

Example 11

Multiplying Rabbits

- Rabbits give birth monthly once they are 3 months old and (let’s assume) they always conceive a single male and female pair.
- You are given a pair of male & female rabbits. Assuming rabbits never die, how many pairs of rabbits do you have after n months?

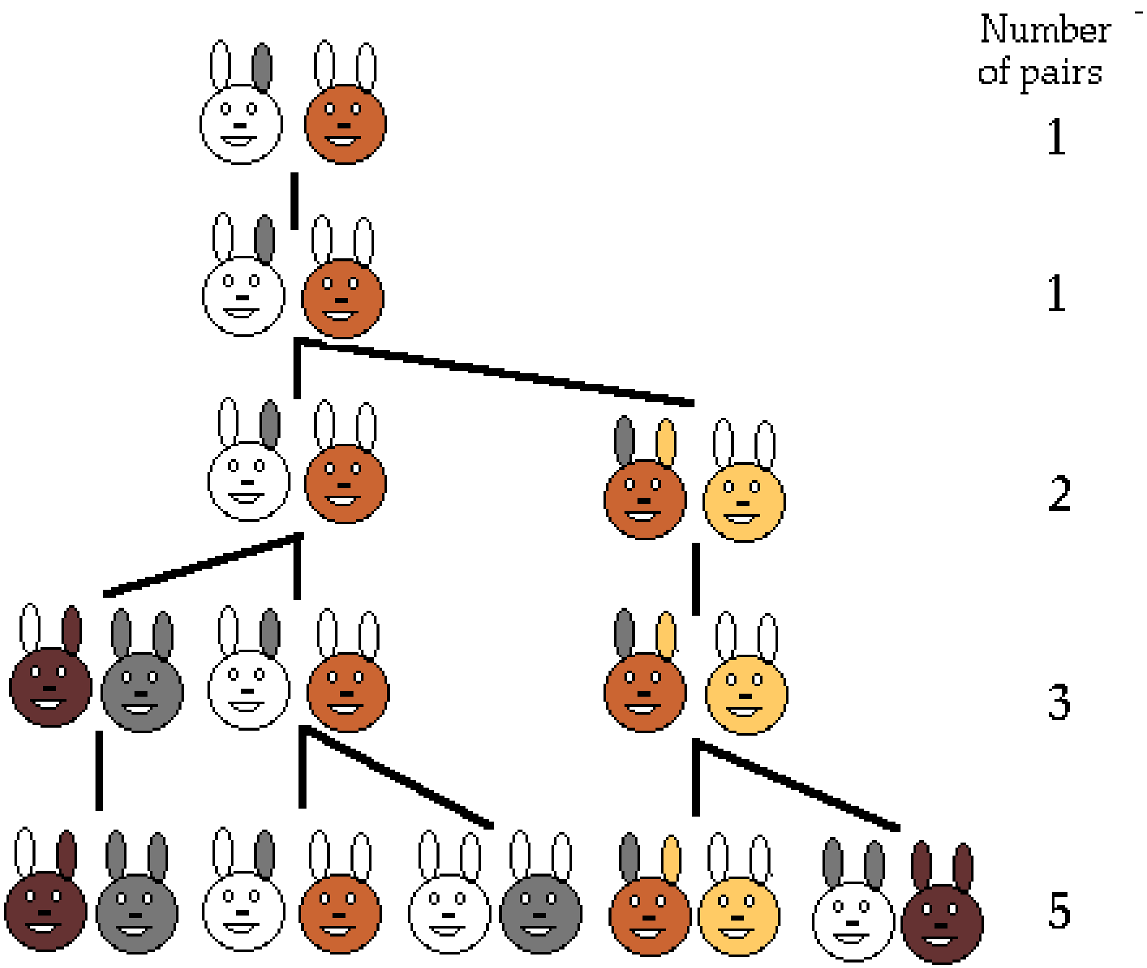
n =	1	2	3	4	5	6	7	8	9	…	n
f(n) =	1	1	2	3	5	8	13	21	34	…	?

Too many!!!

total rabbits = rabbits in previous month + new rabbits
new rabbits in month n = number of rabbits in month n-2

Example 11

Another view of rabbit generations



Example 11

Fibonacci Numbers

- 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:

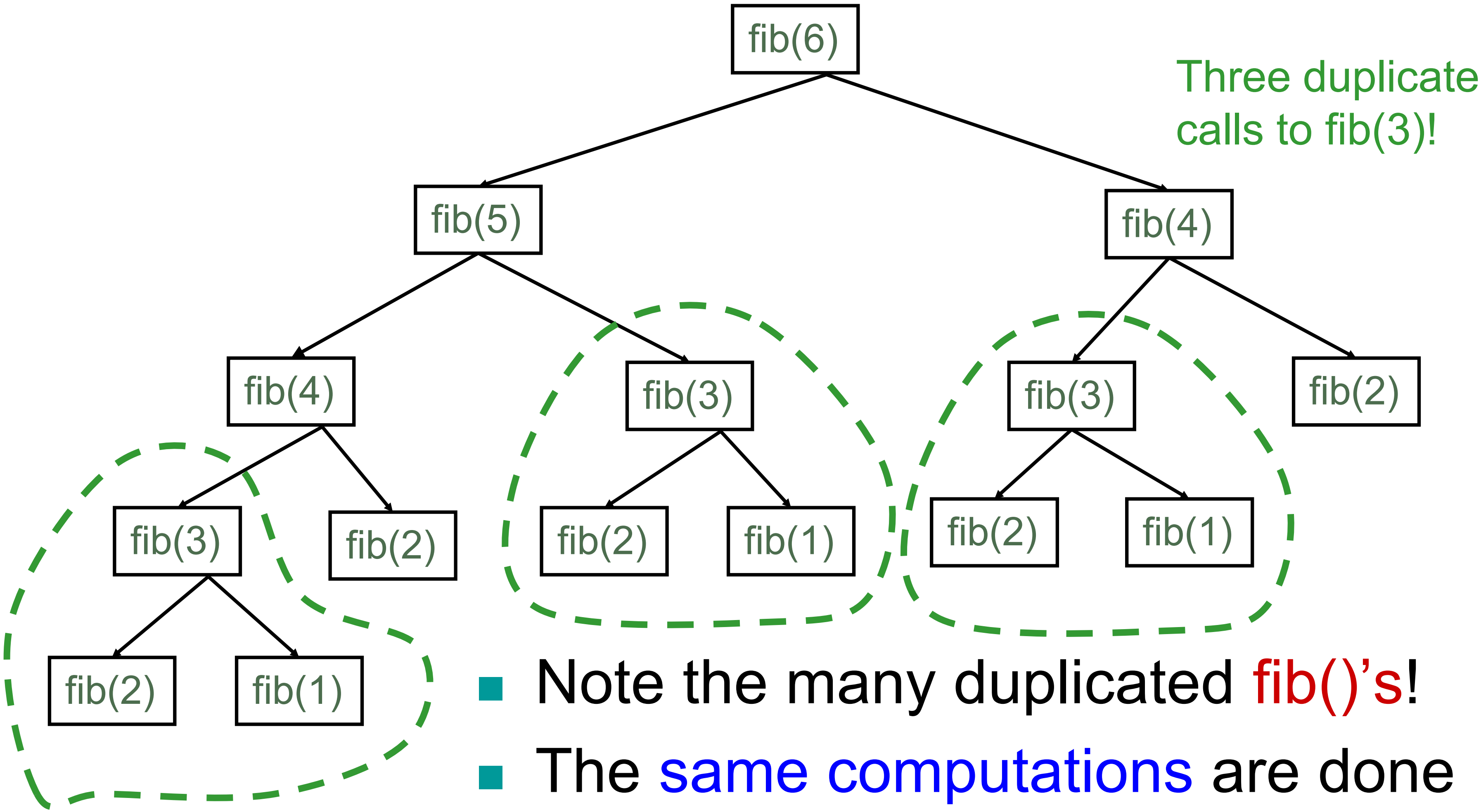
```
public static int fib(int n) {
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Very elegant but extremely inefficient.
Q: Why?

A: Doesn’t reach the base case
B: Repeated work
C: Should put recursive case on top

Example 11

Tracing Fibonacci Calls



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

Example 11

An iterative Fibonacci function

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

Q: Which lines is/are the key to improved efficiency in this implementation?

A: Line A

B: Lines B

C: It's more efficient because it's iterative

Example 12

Find all Permutations of a String

- 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 *tsa* — 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 3 character string e.g. *ast*, we would follow the same procedure.

Example 12

Find all Permutations of a String

```
public class MainClass {
  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);
          permuteString(beginningString + endingString.charAt(i), newString);
        } catch (StringIndexOutOfBoundsException exception) {
          exception.printStackTrace();
        }
      }
  }
}
```

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.

4 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