Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

H. Cecotti

Learning outcomes

- Recursive functions
  - ➢See section 1.5 of the CSci115 book

Introduction

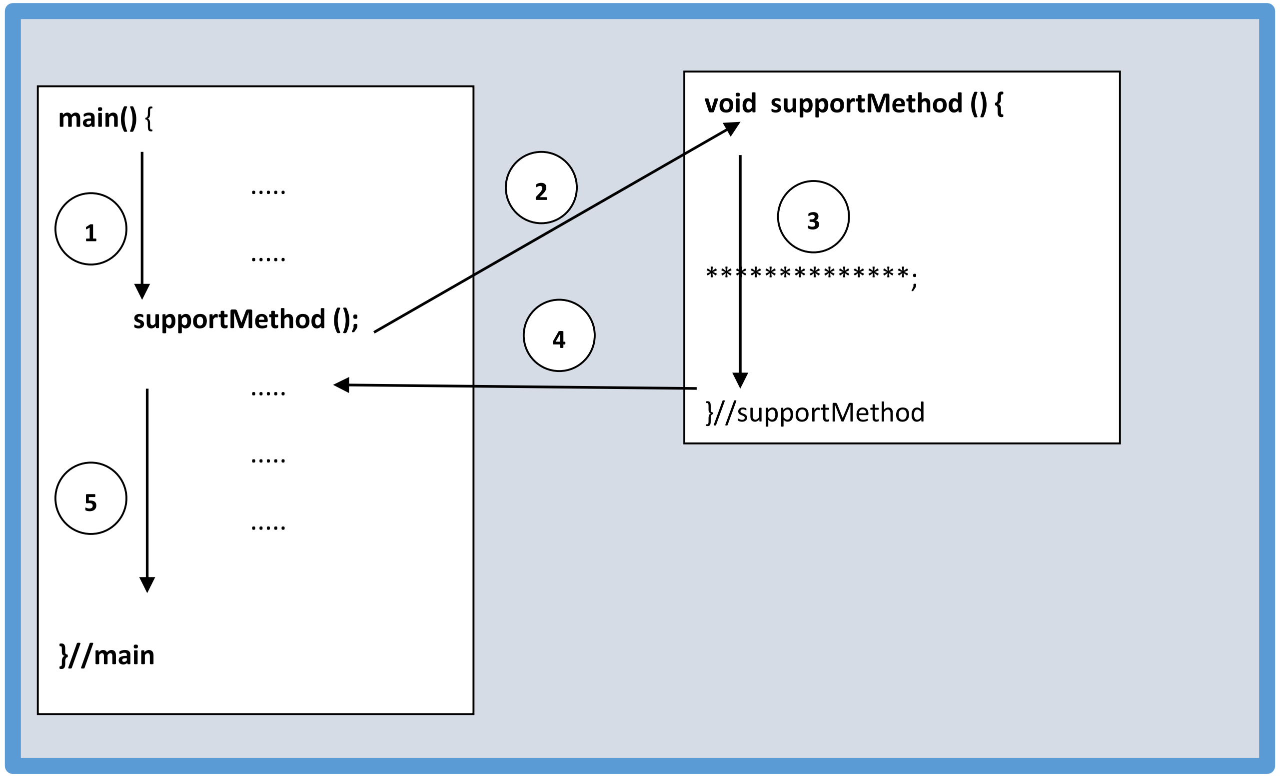
- Recursion in mathematic
  - Recursive definition (induction definition)
  - n: Natural Number
  - P: logical proposition
    - Example: the array is sorted between 1 and n
- What we want
  - **Base case**
    - $P(n_0)$: true
  - **Inductive hypothesis**
    - $P(n)$ true
  - **Inductive step**
    - $P(n)$ true $\rightarrow$ $P(n+1)$ true $\forall\ n>n_0$

Recursion

- Examples: Prove by induction…
  - Summation
    - Show that $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1, \forall n \geq 0$
  - Inequalities
    - Show that that $2^n < n! \; \forall n \geq 4$
  - Divisibility
    - Show that $n^3 - n$ is divisible by 3 $\forall n>0$
- Method
  - Show it is true for the default case (n=0)
  - Show it is true for n+1 by using the fact that is it true at n
    - You must consider the hypothesis !

Method Calling

-



Explanation

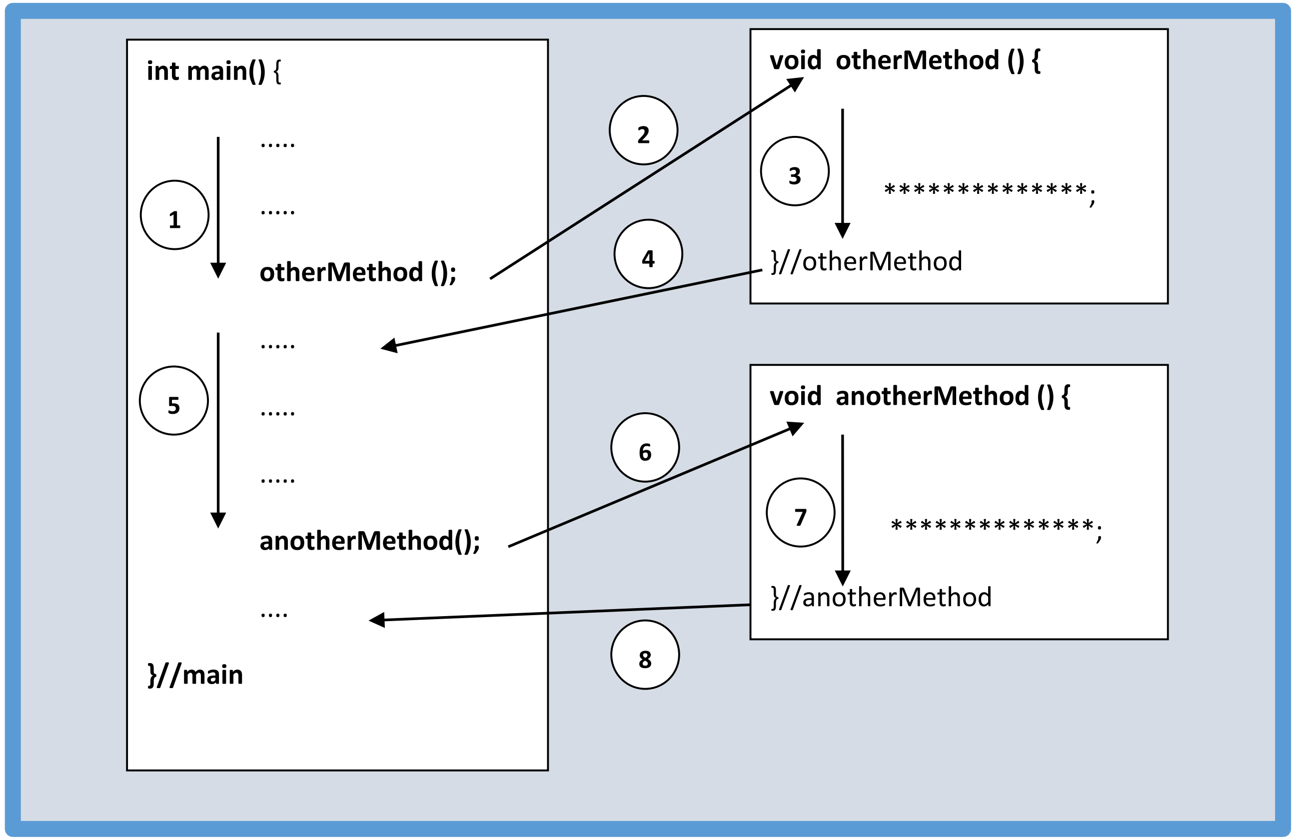
- Control starts with the main( ) method
- Code within the main( ) method
  - It is performed in a “line by line” manner
  - until we reach the line containing a call to perform the method supportMethod( )
- Following the call to perform the method supportMethod( )
  - The control is passed to this method

Explanation

- Code within the **supportMethod( )** method
  - It is performed in a “line by line” manner until we reach the end of the method
- Upon completion of the method **supportMethod()**
  - The control is returned to the main() method to the point immediately following the call
- Code within the main() method is performed in a line by line manner until we reach the end of the main() method

Flow of Control

■



Explanation - 1

- Control starts with the main () method.
- Code within the main() method is performed in a line by line manner, until we reach the call to perform the method otherMethod()
- Following the call to perform the method otherMethod(), control is passed to this method
- Code within the otherMethod() method is performed in a line by line manner until we reach the end of the method

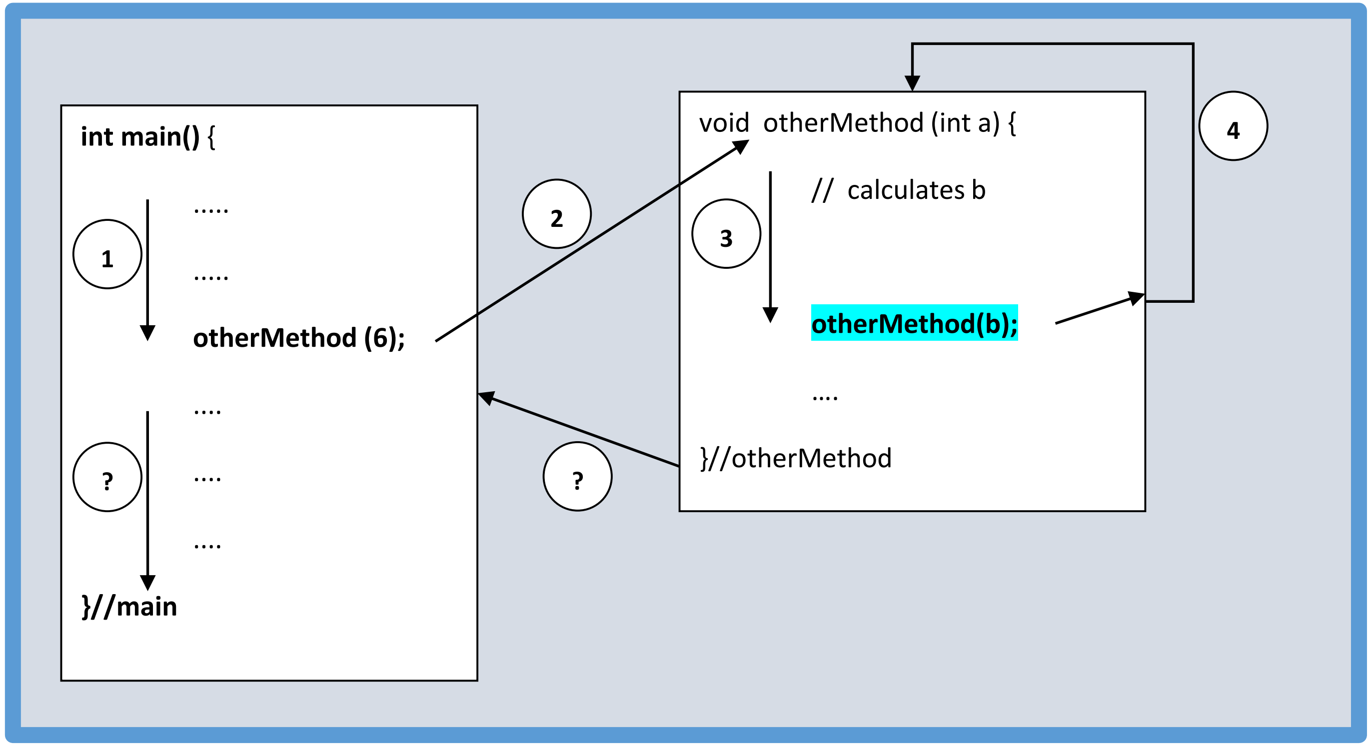
Explanation - 2

- Upon completion of the method otherMethod()
  - control is returned to the point immediately following the call to the method
- Code within the main() method then continues to be performed in a line by line manner until we reach the call to perform the method anotherMethod()
- Following the call to perform the method anotherMethod(), control is passed to this method
- Code within the anotherMethod() method
  - performed in a line by line manner until we reach the end of the method
- Upon completion of the method anotherMethod(),
  - The control is returned to the main() method to the point immediately following the call to the method

Simple Method Calls

- We previously looked at **simple method calls**
- In our example, a method was called within the **main** part of a program
- **Recursion** is when we have a method that contains a line of code that is a further call to the method.
- We say that the method "calls itself"!

Diagrammatically



Explanation - 1

- One of the lines in the method is a **call to itself**

  otherMethod (int a)

- Flow of control is similar to previous **however** when we encounter the line within **otherMethod (int a)** that calls itself,
  - we effectively introduce another occurrence of the method **otherMethod (int a)**

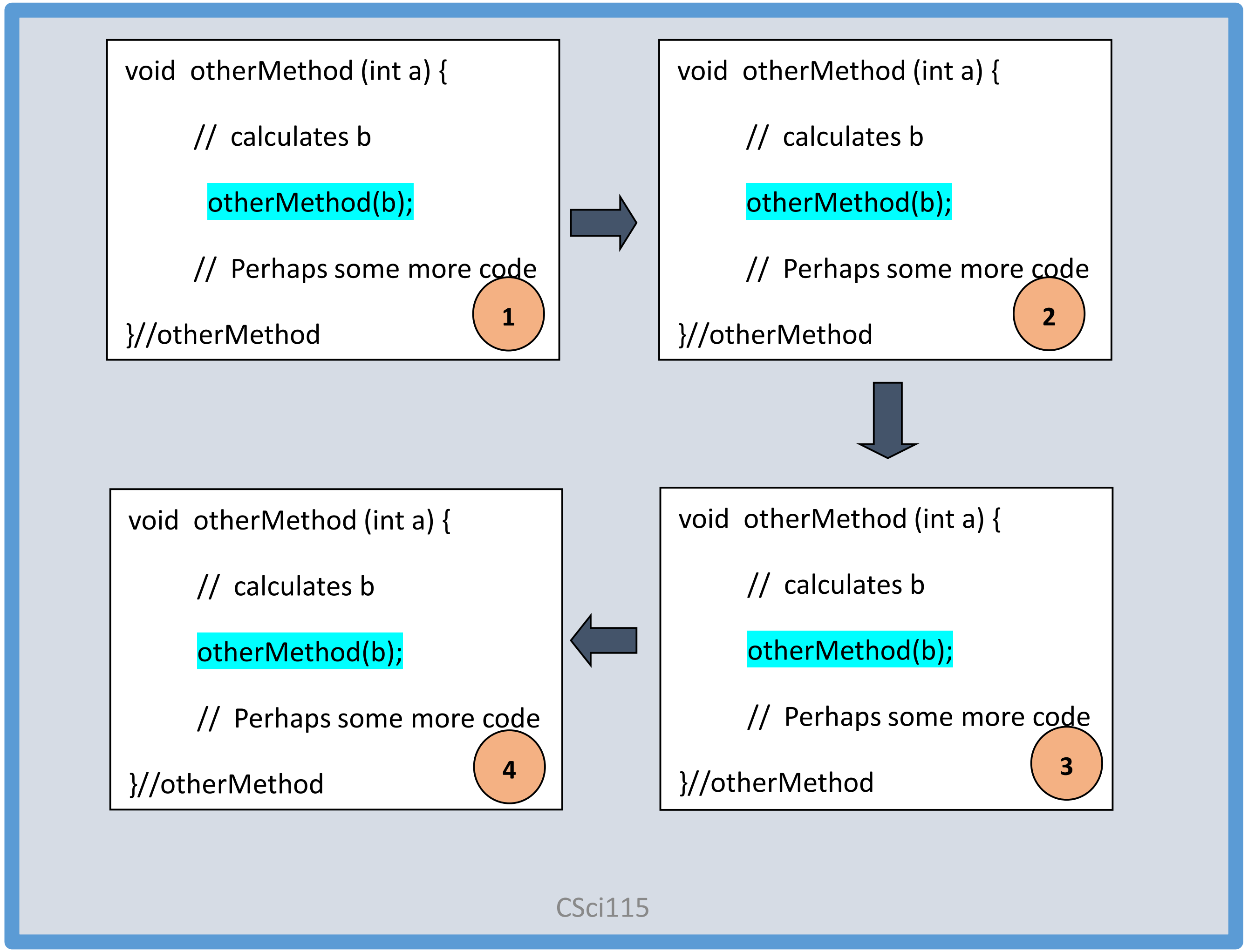
- THE FLOW OF CONTROL IS PASSED TO THE <u>2nd CALL</u> OF THIS METHOD!

Explanation - 2

- This second call to the method is performed
  - Clearly this second call to the method also has a call to the method
  - When this is encountered, FLOW OF CONTROL IS PASSED TO THE **THIRD CALL** TO THIS METHOD
- This third call to the method is performed
  - This third call to the method also has a call to the method
  - When this is encountered, FLOW OF CONTROL IS PASSED TO THE FOURTH CALL TO THIS METHOD
- We could continue in this manner so we might have **numerous** calls to the method
  - We could end up in an infinite loop
    - i.e. we keep on calling the method that calls itself, which calls itself, which calls itself ad infinitum unless there was something that stops further calls to the method

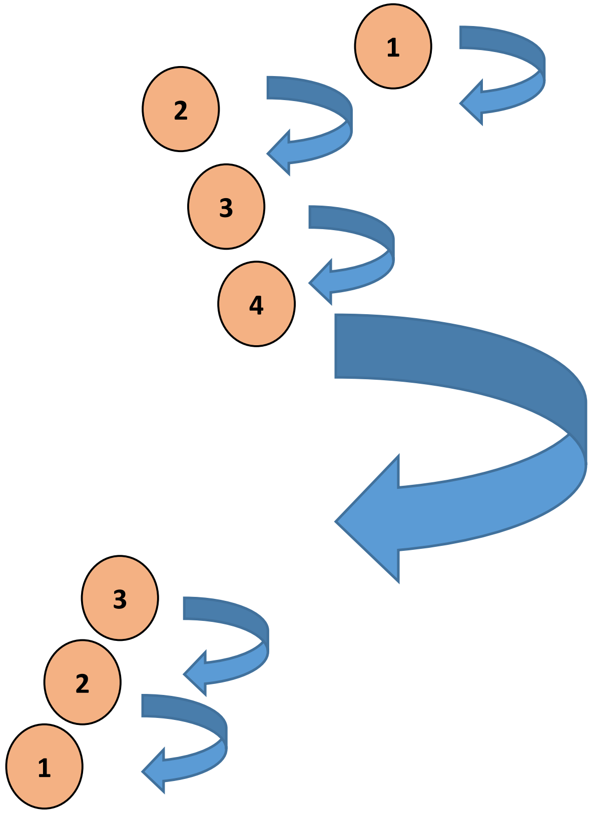
Recursive Calls

-



Order Required

- CALLING ORDER
  - **main** method – call to **otherMethod (int a)**
    - This call to **otherMethod (int a)** calls itself
      - This call to **otherMethod (int a)** calls itself
        - This call to otherMethod (int a) calls itself



- RETURNING ORDER
  - From otherMethod (int a) we
  - return to **otherMethod (int a)** then, we
  - return to **otherMethod (int a)**
  - return to **otherMethod (int a)**
  - return to **main** method

Recursive Methods

- Key points
  - Degree of regularity in the problem with
    - Solutions depend on solutions to smaller instances of the problem(see examples)
- In addition

1. There MUST be a line in the method which is a further call to the method itself
2. This call MUST be a CONDITIONAL CALL

"There must be something that cause the sequence of calls to the method to cease"

WHY? If there is nothing to stop the sequence of calls to the method we will end up with an infinite number of calls to the method!

This is the base case

Example - Factorial of a number

- Factorial of a Number N
  - Written N!
- Expressed as follows:
  - N! = N * (N-1) * (N-2) * (N-3) * .... * 3 * 2 * 1
- Example:
  - 10! = 10 * (9) * (8) * (7) * .... * 3 * 2 * 1

Example - Factorial

- Product of **N** integers
- 0! is not calculated
- 0! is simply defined as being equal to 1

1!	is calculated as →	1	→	**1**
2!	is calculated as →	2 * 1	→	**2**
3!	is calculated as →	3 * 2 * 1	→	**6**
……				
7!	is calculated as	5040	→	(below)
	7 * 6 * 5 * 4 * 3 * 2 * 1		→	5040

Note

- Consider **8!**
  - This is calculated as: 8 * (7 * **6 * 5 * 4 * 3 * 2 * 1**)
- NOW we have just seen that: **7!** is calculated as **(7 * 6 * 5 * 4 * 3 * 2 * 1)**
  - Hence: **8! → 8 * 7!**
  - 8! = 8 * 7!

7! = 7 * 6!

6! = 6 * 5!

5! = 5 * 4!

4! = 4 * 3!

- → **N!** can easily be worked out from **(N-1)!**

Factorial

-

```
void factorial(int number) {
int answer = 1;
if (number < 0)
        cout << "Error - Number has to be positive" << endl;
else {
        for (int loop = number; loop > 1; loop--) {
            answer = answer * loop;
        }
        cout << "Factorial " << number << " = " << answer << endl;
}

int factorial(int number) {
    int answer;
    if (number == 0){
        answer = 1;
    else
        answer = number * factorial (number - 1);
    return answer;
}
```

Other Recursive Methods

- Factorial of an integer N :
  - the PRODUCT of all integers between 1 and N
- You can see that there would be a similar recursive method to calculate the **SUM** of all the integers between 1 and N
- **The Fibonacci Sequence**

  fib (N)=fib (N-1)+ fib(N-2)

  With fib (0) = 0 and fib (1) = 1

Fibonacci Numbers

- A Fibonacci number is calculated as the sum of its 2 Fibonacci predecessors
- **Definitions**
  - fib (0) is defined as 0
  - fib (1) is defined as 1
- **Thereafter:**
  - fib (2) → fib (1) + fib(0) → 1 + 0 → (1)
  - fib (3) → fib (2) + fib(1) → 1 + 1 → (2)
  - fib (4) → fib (3) + fib(2) → 2 + 1 → (3)
  - fib (5) → fib (4) + fib(3) → 3 + 2 → (5)
  - fib (6) → fib (5) + fib(4) → 5 + 3 → (8)

An Example of Linear Recursion: Anagrams

- An anagram is an arrangement of all the letters of a specified word.
- Suppose you want to list all the anagrams of a specified word e.g. rat
- This produces:

rat	rta
art	atr
tra	tar

- The number of possibilities is the **factorial** of the number of letters
  - 3 letters - there are just 6 possible “words”
    - 3x 2 words
  - 4 letters - there are 24 words
    - 4 x 3 x 2 words
  - 5 letters - 120 words

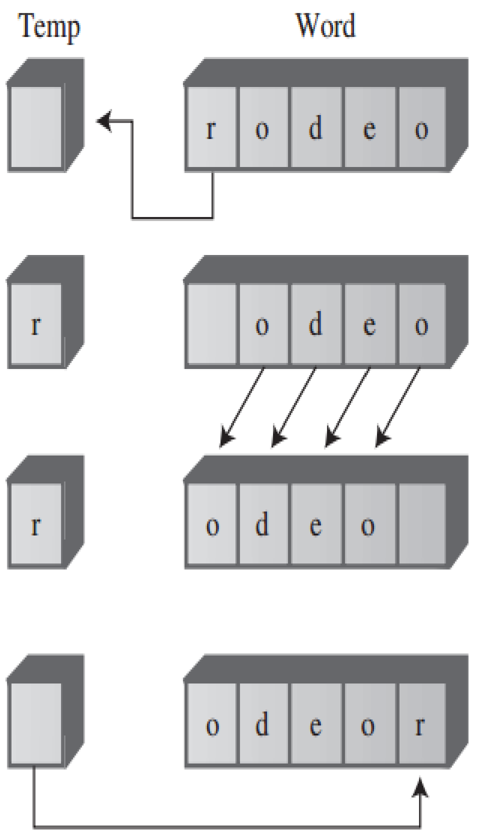
  etc.

Anagrams

- How would you write a program to anagram a word?
- Assume a word has N letters
  - *Anagram* the rightmost N-1 letters
  - *Rotate* all N letters
  - Repeat these steps N times
- To *rotate* the word means to shift

  all the letters one position left,

  except for the leftmost letter, which rotates back to the right
- Rotating the word N times gives each letter a chance to begin the word
  - ➢ See rat example
- While the selected letter occupies the first position,
  - ➢ all other letters are then anagrammed (arranged in every possible position)
- For rat, which has only 3 letters,
  - ➢ anagramming the remaining 2 letters simply switches them



Anagramming the word rat

Word	First Letter	Remaining Letters	Action
rat	r	at	Rotate at
rta	r	ta	Rotate ta
rat	r	at	Rotate rat
atr	a	tr	Rotate tr
art	a	rt	Rotate rt
atr	a	tr	Rotate atr
tra	t	ra	Rotate ra
tar	t	ar	Rotate ar
tra	t	ra	Rotate tra
rat	r	at	Done

Anagrams

- How do we anagram the rightmost (n-1) letters?
  - By invoking a rotate method
- The recursive *doAnagram()*method takes the size of the word to be anagrammed as a parameter
- Each time *doAnagram()* calls itself, it does so with a sequence of characters, one character fewer than before
- The "base case"
  - It occurs when the size of the sequence of characters to be anagrammed is only one character
  - This base case causes the stop!
  - There is no way to rearrange a single character so the method finishes

Anagrams

■

```
//  Method to rotate left all characters from position to end
public static void rotate(char* anyArray, int noOfChars, int newSize) {
   int position = noOfChars - newSize;
   char temp = anyArray[position];
   for (int index = position + 1; index < noOfChars; index++){
            anyArray[index - 1] = anyArray[index];
   }
   anyArray[index-1]=temp;
}

// Recursive method to anagram a word
void doAnagram(char* anyArray, int noOfChars, int newSize) {
    if (newSize > 1) {
        for (int loop = 0; loop < newSize; loop++) {
            doAnagram(anyArray,noOfChars,(newSize-1));
            if (newSize == 2) {
                for (int index = 0; index < noOfChars; index++) {
                   cout << anyArray[index];
                }
                cout << endl;
            }
            rotate(anyArray,noOfChars,newSize);
        }
    }
}

void main() {
        char* myWord = {'R','A','T','S'};
        doAnagram(myWord,4,4);
}
```

Anagrams

Note that every time the

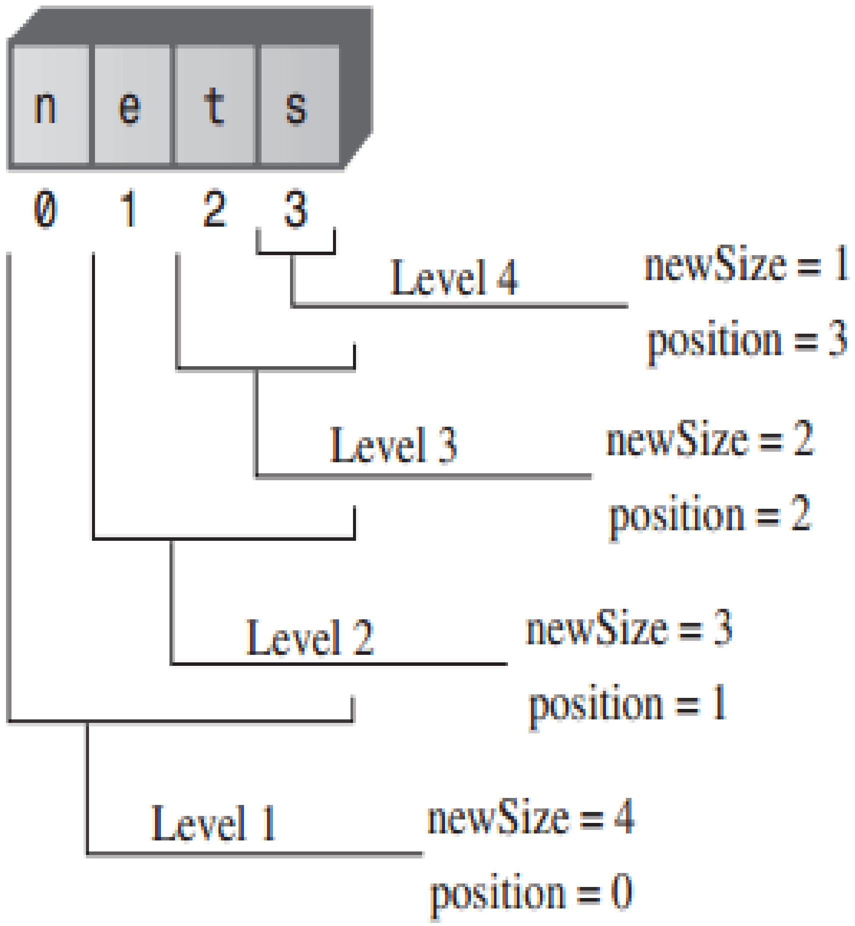
doAnagram()

method calls itself:

(i) the size of the word is one letter smaller

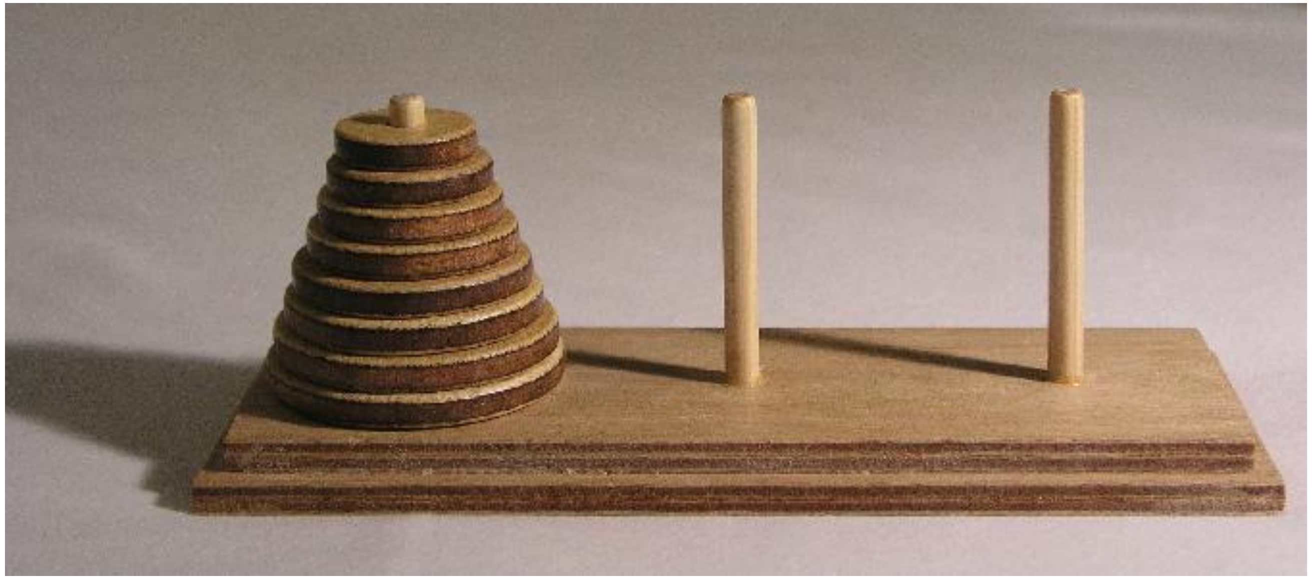
and

(i) the starting position is one cell further to the right:



Hanoi towers

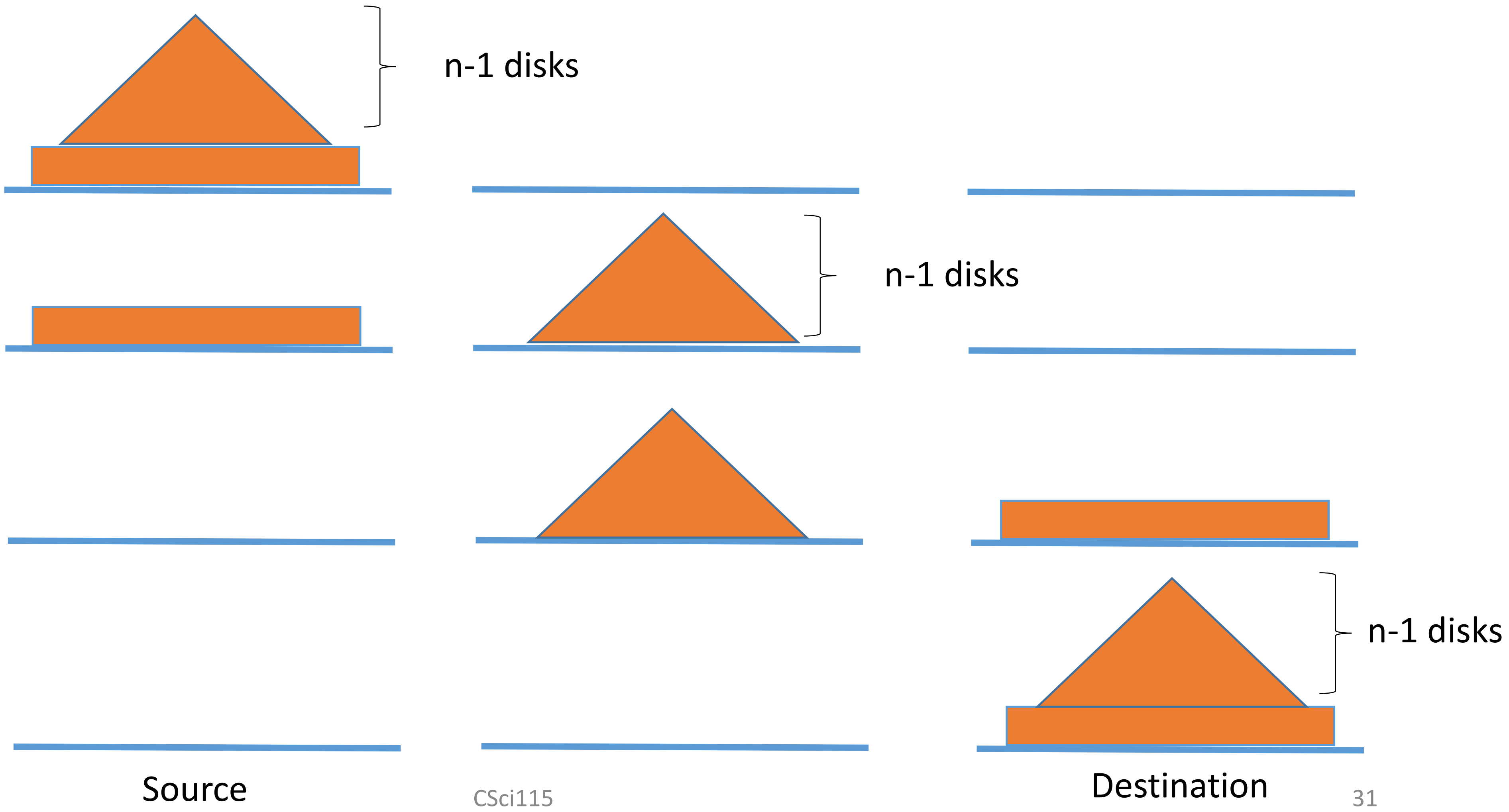
- Move a tower to a different stick…



A B C

Hanoi towers – pseudo code

- The idea



Hanoi towers – pseudo code

- **Base Case**
  - When (n = 1)
    - Move the disc from start pole to end pole
- **Recursive Case**
  - When (n > 1)
    - **Step 1**:
      - Move (n-1) discs from start pole to auxiliary pole.
    - **Step 2**:
      - Move the last disc from start pole to end pole.
    - **Step 3**:
      - Move the (n-1) discs from auxiliary pole to end pole.
    - Steps 1 and 3 are recursive invocations of the same procedure.

Hanoi towers

▪

```
// Hanoi Tower
void solve(int n, String start, String auxiliary, String end) {
       if (n == 1)
           cout << start << " -> " << end << endl;
       else
       {
           solve(n - 1, start, end, auxiliary);
           cout << start << " -> " << end << endl;
           solve(n - 1, auxiliary, start, end);
       }
}

void main(String[] args) {
       int discs = 10;
       solve(discs, "A", "B", "C");
}
```

Binary Search

- **Loop**

```
// Binary Search 1
bool found=false;
while ((low <= high) && (!found)) {
    middle = (low + high) / 2;
    if (search == myArray [middle]) {
            found = true;
    }
    else {
            if (search < myArray [middle]) {
                high = middle - 1;
            }
            else {
                low = middle + 1;
            }
    }
}
```

Recursivity

```
// Binary Search 2
int binarySearch(int* anyArray, int number, int low, int high){
        int middle, answer;
        if (low > high)
            answer=-1;
        else {
            middle = (low + high) / 2;
            if (anyArray[middle] == number)
                answer = middle;
            else {
                if (anyArray [middle] < number)
                    answer = binarySearch(anyArray, number, middle + 1, high);
                else
                    answer = binarySearch(anyArray, number, low, middle - 1);
            }
        }
        return answer;
}
```

Recursive Binary Search

Array being searched

index of first element in the part of the array being searched

```
binarySearch (int [] anyArray, int number,  int low,   int high)
```

value being searched for

index of last element in the part of the array being searched

Recursion can be used to replace the loop

Instead of changing **low** and **high**, the **binarySearch** method is called with new values for **low** and **high**

Recursive Binary Search

- Classic example of:
  - “divide-and-conquer”
- The problem
  - Divided into 2 smaller (sub-)problems
  - Each (sub-)problem is solved separately
- Each smaller sub-problem is divided into 2 smaller sub-sub problems
  - Each sub-sub-problem is solved separately
- Sub-division into increasingly smaller problems continues
  - **until** the base case (smallest problem) is reached
  - i.e. the problem cannot be sub-divided any further

Questions ?

- Reading:
  - ➢See documents on Canvas
  - ➢Section 1.5 in CSci115 book
- Don't forget office hours
  - ➢Come with your laptop and questions