CS1007 lecture #21 notes

tue 16 apr 2002

- news
- recursion
- search
- reading: 11

recursion.

- recursion is defining something in terms of itself
- there are many examples in nature
- and in mathematics
- and in computer graphics, e.g., the Koch snowflake (textbook, p.485)

power function.

- *power* is defined recursively:

$$x^y = \begin{cases} \text{if } y == 0, \;\; x^y = 1 \\ \text{if } y == 1, \;\; x^y = x \\ \text{otherwise}, \;\; x^y = x * x^{y-1} \end{cases}$$

here it is in a Java method.

```
public int power ( int x, int y ) {
   if ( y == 0 ) {
     return( 1 );
   }
   else if ( y == 1 ) {
     return( x );
   }
   else {
     return( x * power( x, y-1 ));
   }
 } // end of power() method
```

- Notice that `power()` calls itself!
- You can do this with any method *except main()*
- BUT beware of infinite loops!!!
- You have to know when and how to stop the recursion — what is the *stopping* condition

let's walk through `power(2,4)`.

	call	x	y	return value
1	power(2,4)	2	4	2 * power(2,3)
2	power(2,3)	2	3	2 * power(2,2)
3	power(2,2)	2	2	2 * power(2,1)
4	power(2,1)	2	1	2

- the first is the *original call*
- followed by three *recursive calls*

stacks.

- the computer uses a data structure called a *stack* to keep track of what is going on
- think of a *stack* like a stack of plates
- you can only take off the top one
- you can only add more plates to the top
- this corresponds to the two basic *stack operations*:
  - *push* — putting something onto the stack
  - *pop* — taking something off of the stack
- when each recursive call is made, power() is pushed onto the stack
- when each return is made, the corresponding power() is popped off of the stack

another example: factorial.

- *factorial* is defined recursively:

$$N! = \begin{cases} \text{if } N == 1, & N! = 1 \\ \text{otherwise}, & N! = N * (N-1)! \end{cases}$$

- (for $N > 0$)

here it is in a Java method.

```
public int factorial ( int N ) {
   if ( N == 1 ) {
     return( 1 );
   }
   else {
     return( N * factorial( N-1 ));
   }
 } // end of factorial() method
```

recursive iteration.

- You can also use recursion to iterate.
- Here's an example. Given:

```
public class ex21a {

  int[] myArray = new int[5];

  public static void main( String[] args ) {
    ex21a ex = new ex21a();
    for ( int i=0; i<5; i++ ) {
      ex.myArray[i] = i + 10;
    }
    ex.printArray( 0 );
  } // end of main() method
```

```
  public void printArray( int index ) {
    if ( index < myArray.length ) {
      System.out.print( myArray[index] + " " );
      printArray( index+1 );
    }
    else {
      System.out.println();
    }
  } // end of printArray() method

} // end of ex21a class
```

normal iteration.

- where normal iteration looks like this:

```
public void printArray() {
  for ( int index=0; index<myArray.length; index++ ) {
    System.out.print( myArray[index] + " " );
  }
  System.out.println();
} // end of printArray() method
```

back to recursive iteration.

- in the recursive version, each call is like one iteration inside the for loop in the iterative version

	call	index	output	next call
1	printArray(0)	0	10	printArray(1)
2	printArray(1)	1	11	printArray(2)
3	printArray(2)	2	12	printArray(3)
4	printArray(3)	3	13	printArray(4)
5	printArray(4)	4	14	printArray(5)
6	printArray(5)	5	newline	— —

more on recursion.

- With recursion, each time the method is invoked, one step is taken towards the resolution of the task the method is meant to complete.
- Before each step is executed, the state of the task being completed is somewhere in the middle of being completed.
- After each step, the state of the task is one step closer to completion.
- In the example above, each time $printArray(i)$ is called, the array is printed from the i-th element to the end of the array.
- In the $power(x, y)$ example, each time the method is called, power is computed for each x^y, in terms of the previous x^{y-1}.
- In the $factorial(N)$ example, each time the method is called, factorial is computed for each N, in terms of the previous $N - 1$.
- The book uses the classic "Towers of Hanoi" example (p477-482). Each time $moveTower()$ is called, one disk is moved from one tower to another. At each point (i.e., at the start of each recursive call), the state of the towers is in the middle of completion, until the final solution is reached.

searching.

- Often, when you have data stored in an array, you need to locate an element within that array.
- This is called searching.
- Typically, you search for a *key* value (simply the value you are looking for) and return its *index* (the location of the value in the array)
- As with sorting, there are many searching algorithms.
- We'll study the following:
  - linear search
    - standard linear search, on sorted or unsorted data
    - modified linear search, on sorted data only
  - binary search
    - iterative binary search, on sorted data only
    - recursive binary search, on sorted data only

linear search on UNSORTED DATA.

- Linear search simply looks through all the elements in the array, one at a time, and stops when it finds the key value.
- This is inefficient, but if the array you are searching is not sorted, then it may be the only practical method.

```
public int linearSearch( int key ) {
  for ( int i=0; i<myArray.length; i++ ) {
    if ( key == myArray[i] ) {
      return( i );
    }
  } // end for i
  return( -1 );
} // end of linearSearch() method
```

linear search on SORTED data.

- If the array you are searching IS sorted, then you can modify the linear search to stop searching if you have looked past the place where the key would be stored if it were in the array.
- This only helps shorten the run time if the key is not in the array...

```
public int modifiedLinearSearch( int key ) {
  for ( int i=0; i<myArray.length; i++ ) {
    if ( key == myArray[i] ) {
      return( i );
    }
    else if ( key < myArray[i] ) {
      return( -1 );
    }
  } // end for i
  return( -1 );
} // end of modifiedLinearSearch() method
```

binary search.

- Binary search is much more efficient than linear search, ON A SORTED ARRAY.
- It CANNOT be used on an unsorted array!
- It takes the strategy of continually dividing the search space into two halves, hence the name *binary*.
- Say you are searching something very large, like the phone book. If you are looking for one name (e.g., “Gilligan”), it is extremely slow and inefficient to start with the A’s and look at each name one at a time, stopping only when you find “Gilligan”. But this is what linear search does.
- Binary search acts much like you’d act if you were looking up “Gilligan” in the phone book.
  - You’d open the book somewhere in the middle, then determine if “Gilligan” appears before or after the page you have opened to.
  - If “Gilligan” appears after the page you’ve selected, then you’d open the book to a later page.

  - If “Gilligan” appears before the page you’ve selected, then you’d open the book to an earlier page.

- You’d repeat this process until you found the entry you are looking for.

binary search, 2.

```
public int binarySearch( int key ) {
  int lo = 0, hi = myArray.length-1, mid;
  while ( lo <= hi ) {
    mid = ( lo + hi ) / 2;
    if ( key == myArray[mid] ) {
      return( mid );
    }
    else if ( key < myArray[mid] ) {
      hi = mid - 1;
    }
    else {
      lo = mid + 1;
    }
  } // end while
  return( -1 );
} // end of binarySearch() method
```

recursive binary search.

```
public int recursiveBinarySearch( int key, int lo, int hi ) {
  if ( lo <= hi ) {
    int mid = ( lo + hi ) / 2;
    if ( key == myArray[mid] ) {
      return( mid );
    }
    else if ( key < myArray[mid] ) {
      return( recursiveBinarySearch( key, lo, mid-1 ));
    }
    else {
      return( recursiveBinarySearch( key, mid+1, hi ));
    }
  }
  else {
    return( -1 );
  }
} // end of recursiveBinarySearch() method
```