Chapter 13

Recursion

Topics

- Simple Recursion
- Recursion with a Return Value
- Binary Search Revisited
- Recursion Versus Iteration

Simple Recursion

- When solving a problem using recursion, the idea is to transform a big problem into a smaller, similar problem.
- Eventually, as this process repeats itself and the size of the problem is reduced at each step, we will arrive at a very small, **easy-to-solve** problem.
- That easy-to-solve problem is called the **base case**.
- The formula that reduces the size of a problem is called the **general case**.

Recursive Methods

- A recursive method calls itself, i.e. in the body of the method, there is a call to the method itself.
- The arguments passed to the recursive call are smaller in value than the original arguments.

Simple Recursion

- When designing a recursive solution for a problem, we need to do two things:
  - Define the base case.
  - Define the rule for the general case.

Printing "Hello World" *n* Times Using Recursion

- In order to print "Hello World" *n* times (*n* is greater than 0), we can do the following:
  - Print "Hello World"
  - Print "Hello World" (*n – 1*) times
- This is the general case.
- We have reduced the size of the problem from size *n* to size (*n – 1*).

Printing "Hello World" *n* Times Using Recursion

- Printing "Hello World" (*n – 1*) times will be done by
  - Printing "Hello World"
  - Printing "Hello World" (*n – 2*) times
- … and so on
- Eventually, we will arrive at printing "Hello World" 0 times: that is easy to solve; we do nothing. That is the base case.

Coding the Recursive Method

```
public static void printHelloWorldNTimes( int n )
{
  if ( n > 0 )
  {
    System.out.println( "Hello World" );
    printHelloWorldNTimes( n - 1 );
  }
  // if n is 0 or less, do nothing
}
```

- *See Example 13.1 RecursiveHelloWorld.java*

Recursion with a Return Value

- A recursive method is a method; as such, it can be a value-returning method.
- In a value-returning method, the *return* statement can include a call to another value-returning method.
- For example,

```
public int multiplyAbsoluteValueBy3( int n )
{
   return ( 3 * Math.abs( n ) );
}
```

Recursion with a Return Value

- In a recursive value-returning method, the *return* statement can include a call to the method itself.
- The return value of a recursive value-returning method often consists of an expression that includes a call to the method itself:

```
return ( expression including a
         recursive call to the method );
```

Factorial

- The factorial of a positive number is defined as

```
factorial( n ) = n!
    = n * ( n – 1 ) * ( n – 2 ) * … 3 * 2 * 1
```

  – By convention,

```
factorial( 0 ) = 0! = 1
```

  – The factorial of a negative number is not defined.
- Can we find a relationship between the problem at hand and a smaller, similar problem?

Factorial

```
factorial( n ) = n!
   = n * ( n – 1 ) * ( n – 2 ) * … 3 * 2 * 1
factorial( n – 1 ) = ( n – 1 )!
   = ( n – 1 ) * ( n – 2 ) * … 3 * 2 * 1
```

- So we can write

```
factorial( n ) = n * factorial( n – 1 )
```

- That formula defines the general case.

Factorial

```
factorial( n ) = n * factorial( n – 1 )
```

- At each step, the size of the problem is reduced by 1: we progress from a problem of size *n* to a problem of size (*n – 1*)
- A call to *factorial(n)* will generate a call to *factorial(n – 1)*, which in turn will generate a call to *factorial(n – 2)*, ….
- Eventually, a call to *factorial(0)* will be generated; this is our easy-to-solve problem. We know that *factorial(0) = 1*. That is the base case.

Code for a Recursive Factorial Method

```
public static int factorial( int n )
{
  if ( n <= 0 ) // base case
     return 1;
  else          // general case
     return ( n * factorial( n – 1 ) );
}
```

- *See Example 13.2 RecursiveFactorial.java*

Common Error Trap

When coding a recursive method, failure to code the base case will result in a run-time error.

If the base case is not coded, when the method is called, the recursive calls keep being made because the base case is never reached.

This eventually generates a ***StackOverflowError***.

Recursion Versus Iteration

- A recursive method is implemented using decision constructs (*if/else* statements) and calls itself.
- An iterative method is implemented with looping constructs (*while* or *for* statements) and repeatedly executes the loop.
- Printing "Hello World" *n* times and calculating a factorial can easily be coded using iteration.
- Other problems, such as the Towers of Hanoi and Binary Search, are more easily coded using recursion.

Recursion Versus Iteration

- Considerations when deciding to use recursion or iteration include:
  - Efficiency of the method at execution time: often, recursion is slower due to overhead associated with method calls.
  - Readability and maintenance: often, a recursive formulation is easier to read and understand than its iterative equivalent.

Back Up Slides

Recursive Binary Search

- Our Recursive Binary Search will implement a Binary Search using recursion.
- Review: A Binary Search searches a sorted array for a search key value.
  - If the search key is found, we return the index of the element with that value.
  - If the search key is not found, we return -1.

The Binary Search Algorithm

- We begin by comparing the middle element of the array with the search key.
- If they are equal, we found the search key and return the index of the middle element.
- That is a base case.

Binary Search (con't)

- If the middle element's value is greater than the search key, then the search key cannot be found in elements with higher array indexes. So, we continue our search in the left half of the array.
- If the middle element's value is less than the search key, then the search key cannot be found in elements with lower array indexes lower. So, we continue our search in the right half of the array.

The Binary Search Algorithm (con't)

- Searching the left or right subarray is made via a recursive call to our Binary Search method. We pass the subarray to search as an argument. Since the subarray is smaller than the original array, we have progressed from a bigger problem to a smaller problem. That is our general case.
- If the subarray to search is empty, we will not find our search key in the subarray, and we return –1. That is another base case.

Example of a Recursive Binary Search

- For example, we will search for the value 7 in this sorted array:

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- To begin, we find the index of the center element, which is 8, and we compare our search key (7) with the value 45.

Recursive Binary Search Example (con't)

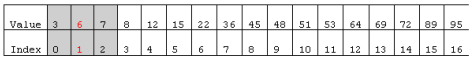
- A recursive call is made to search the left subarray, comprised of the array elements with indexes between 0 and 7 included.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 3, so we compare 7 to the value 8.

Recursive Binary Search Example (con't)

- A recursive call is made to search the left subarray, comprised of the array elements with indexes between 0 and 2 included.

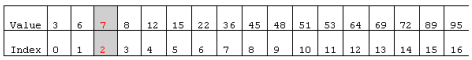


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 1, so we compare 7 to the value 6.

Binary Search: Finding the Search Key

- A recursive call is made to search the right subarray, comprised of the only array element with index between 2 and 2 included.

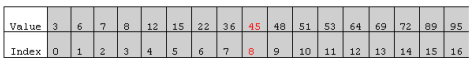


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The value of the element at index 2 matches the search key, so we have reached a base case and we return the index 2 (successful search).

Recursive Binary Search Example 2

- This time, we search for a value not found in the array, 34. Again, we start with the entire array and find the index of the middle element, which is 8.

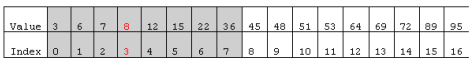


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- We compare our search key (34) with the value 45.

Recursive Binary Search Example 2 (con't)

- A recursive call is made to search the left subarray, comprised of the array elements with indexes between 0 and 7 included.

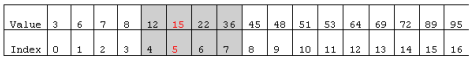


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 3, so we compare 34 to the value 8.

Recursive Binary Search Example 2 (con't)

- A recursive call is made to search the right subarray, comprised of the array elements with indexes between 4 and 7 included.

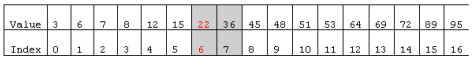


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 5, so we compare 34 to the value 15.

Recursive Binary Search Example 2 (con't)

- A recursive call is made to search the right subarray, comprised of the array elements with indexes between 6 and 7 included.

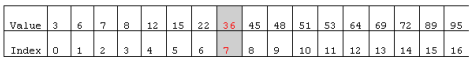


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 6, so we compare 34 to the value 22.

Recursive Binary Search 2: Search Key is Not Found

- A recursive call is made to search the right subarray, comprised of the only array element with index between 7 and 7 included.



Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center (only) element is now 7, so we compare 34 to the value 36.
- A recursive call is made to search the left subarray, but that left subarray is empty. We have reached the other base case, and we return –1, indicating an unsuccessful search.

Recursive Binary Search Method

- How many and what parameters does our recursive binary search method take?
- Our non-recursive method, in Chapter 8, took only two parameters: the array and the search key. The subarray being searched is defined inside the method by two local variables, *start* and *end*, representing the indexes of the first and last elements of the subarray

Recursive Binary Search Method

- When we make a recursive call to search the left or the right subarray, our recursive call will define the subarray that we search.
- Thus, our recursive Binary Search method will have two extra parameters, representing the indexes of the first and last elements of the subarray to search.

Binary Search Code

```
public int recursiveBinarySearch
   ( int [] arr, int key, int start, int end )
{
   if ( start <= end )
   {
      int middle = ( start + end ) / 2;
      if ( arr[middle] == key ) // key found
         return middle;   // one base case
      else if ( arr[middle] > key ) // look left
         return recursiveBinarySearch
            ( arr, key, start, middle - 1 );
      else                          // look right
         return recursiveBinarySearch
            ( arr, key, middle + 1, end );
   }
   else     // key not found
      return -1;  // another base case
}
```

Recursion with Two Base Cases

- Complex recursive formulations can involve more than one recursive call, with each call being made with different arguments.
- This, in turn, means that we can have more than one base case.

Combinations

- How many different ways can we choose *p* players among *n* players?
- We assume that *n* is greater than or equal to *p* (otherwise the answer is 0).
- Call that number *Combinations(n, p)*.

- We want to come up with a recursive solution to this problem.

Analyzing the *Combinations* Problem

- Let's consider some easy cases:
- If *p* is equal to *n*, we have no choice but to pick all the players; there is only one way to do that.

  So *Combinations(n, n) = 1*.
- If *p* is equal to 0, then we do not pick any players; there is only one way to do that.

  So *Combinations(n, 0) = 1*.
- What is the answer in the general case?

  Combinations(n, p) = ?

Analyzing the *Combinations* Problem

- Let's focus on one player, call him Louis.
- We can either pick Louis or not pick Louis, and these two options are mutually exclusive.
- So *Combinations(n, p) =*

```
number of different ways of picking p players
among n, picking Louis.
        +
number of different ways of picking p players
among n, not picking Louis.
```

Analyzing the *Combinations* Problem

- If we pick Louis, we will have to pick *(p – 1)* more players among *(n – 1)* players (we cannot pick Louis twice). That number, by definition, is *Combinations(n – 1, p – 1)*.
- If we do not pick Louis, we will have to pick *p* players among *(n – 1)* players (we do not pick Louis). That number, by definition, is *Combinations(n – 1, p)*.

Combinations: The General Case

- Therefore,

  Combinations(n, p) =

  Combinations(n - 1, p - 1)

  +

  Combinations(n - 1, p)
- That is our formula for the general case. Note that we are progressing from one large problem to two smaller problems.

Defining the First Base Case

- Consider the first term of the right side of the equation, *Combinations(n - 1, p - 1)*
- Both parameters, *n* and *p*, decrease by 1.
- Since *n* is greater than or equal to *p*, eventually *p* will reach 0.
- We know that *Combinations(n, 0) = 1*.
- That is our first base case.

Defining the Second Base Case

- Consider the second term of the right side of the equation, *Combinations(n - 1, p)*
- The first parameter, *n*, decreases by 1, whereas *p* is unchanged.
- Since *n* is greater than or equal to *p*, eventually *n* will reach *p*.
- We know that *Combinations(n, n) = 1*.
- That is our second base case.

Combinations Code

```
public static int combinations( int n, int p )
{
  if ( p == 0 )       // base case # 1
    return 1;
  else if ( n == p ) // base case # 2
    return 1;
  else                // general case
    return ( combinations( n - 1, p - 1 )
             + combinations( n - 1, p ) );
}
```

- *See Example 13.6 RecursiveCombinations.java*

Greatest Common Divisor

- The Greatest Common Divisor (gcd) of two numbers is the greatest positive integer that divides evenly into both numbers.
- The Euclidian algorithm finds the gcd of two positive numbers *a* and *b*.
  - It is based on the fact that:

    `gcd( a, b ) = gcd ( b, remainder of a / b )`

    (assuming *a* is greater than *b* and *b* is different from 0)

GCD: Euclidian Algorithm

Step 1:

```
r0 = a % b
if ( r0 is equal to 0 )
    gcd( a, b ) = b
    stop
else
    go to Step 2
```

Step 2:

```
Repeat Step 1 with b and r0, instead of a and b.
```

GCD Example: Euclidian Algorithm

If *a* = 123450 and *b* = 60378, then …

60378 % 123450 = 2694 (different from 0)
60378 % 2694 = 1110 (different from 0)
2694 % 1110 = 474 (different from 0)
1110 % 474 = 162 (different from 0)
474 % 162 = 150 (different from 0)
162 % 150 = 12 (different from 0)
150 % 12 = 6 (different from 0)
12 % 6 = 0
➔ gcd(123450, 60378) = 6

GCD Code

```
public static int gcd( int dividend, int divisor )
{
  if ( dividend % divisor ==  0 )
    return divisor;
  else          // general case
    return ( gcd( divisor, dividend % divisor ) );
}
```

- *See Example 13.4 RecursiveGCD.java*

Animation Using Recursion

- We can use recursion to move an object on the screen from one location to another.
- A recursive formulation for the general case is:
  1. move the object one pixel.
  2. move the object the rest of the distance.

Animation Using Recursion

- At every step, the distance to move the object decreases by 1.
- Eventually, the distance reaches 0, in which case we do nothing. That is the base case.

- *See Example 13.11 AstronautClient.java*