Recursion

Weiss Ch. 7, pp294-349

Recursive thinking

- *Recursion* is a programming technique in which a method can call itself to solve a problem
- A *recursive definition* is one which uses the word or concept being defined in the definition itself
- In some situations, a recursive definition can be an appropriate way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively

Recursive definitions

- Consider the following list of numbers:
  `24 -> 88 -> 40 -> 37 /`

- A list can be defined recursively

```
Either   LIST = null /
  or     LIST = element -> LIST
```

- That is, a LIST is defined to be either empty (null), or an element followed by a LIST
  - (The concept of a LIST is used to define itself)
  - How would we confirm that null is a LIST? That one element is a LIST? That three elements are a LIST?

More recursive definitions

- An arithmetic expression is defined as:
  - a numeric constant
  - an numeric identifier
  - an *arithmetic expression* enclosed in parentheses
  - 2 *arithmetic expressions* with a binary operator like + - / * %
- Note: The term arithmetic expression is defined by using the term arithmetic expression!
  - (not the first two bullets)

Recursive algorithms

- A *recursive algorithm* is one that refers to itself

- Show everything in a folder and all it subfolders:
  1. show everything in top folder
  2. show everything in each subfolder in the same manner

- Look up a word in a dictionary:
  1. look up a word (use alphabetical ordering) or
  2. look up word to define the word you are looking up

Factorial example

- The *factorial* for any positive integer N, written N!, is defined to be the product of all integers between 1 and N inclusive

$$\boxed{n! = n \times (n-1) \times (n-2) \times \ldots \times 1}$$

```
public static int factorial(int n) {
   int product = 1;
   for (int i = 1; i <= n; i++)
    product *= i;
   return product;
}
```

Exercise: trace execution (show method calls) for n=5

Recursive factorial

- *factorial* can also be defined recursively:

$$
\boxed{f(n) = \begin{cases} n \geq 1 \Rightarrow n \times f(n-1) \\ n = 0 \Rightarrow 1 \end{cases}}
$$

- A factorial is defined in terms of another factorial until the basic case of 0! is reached

```
public static int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Recursive programming

- A method in Java can call itself; if written that way, it is called a *recursive method*
- A recursive method solves some problem. The code of a recursive method should be written to handle the problem in one of two ways:
  - **Base case**: a simple case of the problem that can be answered directly; does not use recursion.
  - **Recursive case**: a more complicated case of the problem, that isn't easy to answer directly, but can be expressed elegantly with recursion; makes a recursive call to help compute the overall answer

Recursive power example

- Write method `pow` that takes integers `x` and `y` as parameters and returns x^y.
  $x^y = x * x * x * \ldots * x$ (y times, in total)

- An iterative solution:

```
public static int pow(int x, int y) {
  int product = 1;
  for (int i = 0; i < y; i++)
    product = product * x;
  return product;
}
```

Recursive power function

- Another way to define the power function:

pow(x, 0) = 1
pow(x, y) = x * pow(x, y-1),   y > 0

```
public static int pow(int x, int y) {
  if (y == 0)
    return 1;
  else
    return x * pow(x, y - 1);
}
```

Why Do Recursive Methods Work?

Activation Records on the **Run-time Stack** are the key:

- Each time you call a function (any function) you get a new activation record.
- Each activation record contains a copy of all local variables and parameters for that invocation.
- The activation record remains on the stack until the function returns, then it is destroyed.

Try yourself: use your IDE's debugger and put a breakpoint in the recursive algorithm

Look at the call-stack.

How recursion works

- each call sets up a new instance of all the parameters and the local variables
- as always, when the method completes, control returns to the method that invoked it (which might be another invocation of the same method)

```
pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 1
          = 64
```

Activation records

- **activation record**: memory that Java allocates to store information about each running method
  - return point ("RP"), argument values, local variable values
  - Java stacks up the records as methods are called; a method's activation record exists until it returns
  - drawing the act. records helps us *trace* the behavior of a recursive method

```
 ____________________________________
| x  = [ 4 ]        y = [ 0 ] | pow(4, 0)
| RP = [pow(4,1)]             |
|______________________________|
| x  = [ 4 ]        y = [ 1 ] | pow(4, 1)
| RP = [pow(4,2)]             |
|______________________________|
| x  = [ 4 ]        y = [ 2 ] | pow(4, 2)
| RP = [pow(4,3)]             |
|______________________________|
| x  = [ 4 ]        y = [ 3 ] | pow(4, 3)
| RP = [main]                 |
|______________________________|
|                             | main
|______________________________|
```

Infinite recursion

- a definition with a missing or badly written base case causes **infinite recursion**, similar to an infinite loop
  - avoided by making sure that the recursive call gets closer to the solution (moving toward the base case)

```
public static int pow(int x, int y) {
  return x * pow(x, y - 1);  // Oops!  Forgot base case
}

pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 4 * pow(4, -1)
          = 4 * 4 * 4 * 4 * 4 * pow(4, -2)
          = ... crashes: Stack Overflow Error!
```

Another Example: Broken Recursive Factorial

```
public static int Brokenfactorial(int n){
   int x = Brokenfactorial(n-1);
   if (n == 1)
    return 1;
   else
    return n * x;
}
```

- What's wrong here? Trace calls "by hand"
  - BrFact(2) -> BrFact(1) -> BrFact(0) -> BrFact(-1) -> BrFact(-2) -> …
  - Problem: we do the recursive call first before checking for the base case
  - Never stops! Like an infinite loop!

Tracing recursive methods

Consider the following method:

```
public static int mystery1(int x, int y)
   {
   if (x < y)
      return x;
   else
      return mystery1(x - y, y);
}
```

For each call below, indicate what value is returned:

```
mystery1(6, 13)       _______________
mystery1(14, 10)      _______________
mystery1(37, 10)      _______________
mystery1(8, 2)        _______________
mystery1(50, 7)       _______________
```

Tracing recursive methods

```
public static void mystery2(int n) {
  if (n <= 1)
    System.out.print(n);
  else {
    mystery2(n/2);
    System.out.print(", " + n);
  }
}
```

For each call below, indicate what output is printed:

```
mystery2(1)         ____________
mystery2(2)         ____________
mystery2(3)         ____________
mystery2(4)         ____________
mystery2(16)        ____________
mystery2(30)        ____________
mystery2(100)       ____________
```

Tracing recursive methods

```
public static int mystery3(int n) {
  if (n < 0)
    return -mystery3(-n);
  else if (n < 10)
    return n;
  else
    return mystery3(n/10 + n % 10);
}
```

For each call below, indicate what value is returned:

```
mystery3(6)          _______________
mystery3(17)         _______________
mystery3(259)        _______________
mystery3(977)        _______________
mystery3(-479)       _______________
```

Tracing recursive methods

```
public static void mystery4(String s) {
  if (s.length() > 0) {
    System.out.print(s.charAt(0));
    if (s.length() % 2 == 0)
      mystery4(s.substring(0, s.length() - 1));
    else
      mystery4(s.substring(1, s.length()));
    System.out.print(s.charAt(s.length() - 1));
  }
}
```

For each call below, indicate what output is printed:

```
mystery4("")                    _______________
mystery4("a")                   _______________
mystery4("ab")                  _______________
mystery4("bc")                  _______________
mystery4("abcd")                _______________
```

Recursive Design

Recursive methods/functions **require**:

1) One or more (non-recursive) **base cases** that will cause the recursion to end.

```
if (n == 1) return 1;
```

2) One or more **recursive cases** that operate on smaller problems **and** get you closer to the base case.

```
return n * factorial(n-1);
```

Note: The base case(s) should always be checked before the recursive call.

Rules for Recursive Algorithms

- Base case - must have a way to end the recursion
- Recursive call - must *<u>change</u>* at least one of the parameters *<u>and</u>* make progress towards the base case
  - – exponentiation (x,y)
    - base: if y is 0 then return 1
    - recursive: else return (multiply x times exponentiation(x,*y-1*))

How to Think/Design with Recursion

- Many people have a hard time writing recursive algorithms
- The key: focus **<u>only</u>** at the current “stage” of the recursion
  - Handle the base case, then…
  - Decide what recursive-calls need to be made
    - **<u>Assume they work (as if by magic)</u>**
  - Determine how to use these calls’ results

Example: List Processing

- Is an item in a list? First, get a reference *current* to the first node
  - (Base case) If *current* is null, return false
  - (Base case #2) If the first item equals the target, return true
  - (Recursive case – might be on the remainder of the list)
    - current = current.next
    - return result of recursive call on new current

Recursive numeric problems

- *Problem*: Given a decimal integer *n* and a base *b*, print *n* in base *b*.
  (Hint: consider the / and % operators to divide *n*.)

- *Problem*: Given integers *a* and *b* where *a* >= *b*, find their greatest common divisor ("GCD"), which is the largest number that is a factor of both *a* and *b*. Use Euclid's formula, which states that:

  GCD(a, b) = GCD(b, a MOD b)

  (Hint: What should the base case be?)

Recursive printing problem

- *Problem*: Write a method `starString` that takes an integer n as an argument and returns a string of stars (asterisks) 2^n long (i.e., 2 to the nth power). For example:

starString(0) should return "*" (because 2^0 == 1)

starString(1) should return "**" (because 2^1 == 2)

starString(2) should return "****" (because 2^2 == 4)

starString(3) should return "********" (because 2^3 == 8)

starString(4) should return "****************" (2^4 == 16)

Recursive string problems

- *Problem*: Write a recursive method `isPalindrome` that takes a string and returns whether the string is the same forwards as backwards.
(Hint: examine the end letters.)

- *Problem*: Write a recursive method `areAnagrams` that takes two strings *w1* and *w2* and returns whether they are anagrams of each other; that is, whether the letters of *w1* can be rearranged to form the word *w2*.

Searching and recursion

- Problem: Given a <u>sorted</u> array *a* of integers and an integer *i*, find the index of any occurrence of *i* if it appears in the array. If not, return -1.
  - We could solve this problem using a standard iterative search; starting at the beginning, and looking at each element until we find *i*
  - What is the runtime of an iterative search?
- However, in this case, the array is sorted, so does that help us solve this problem more intelligently? Can recursion also help us?

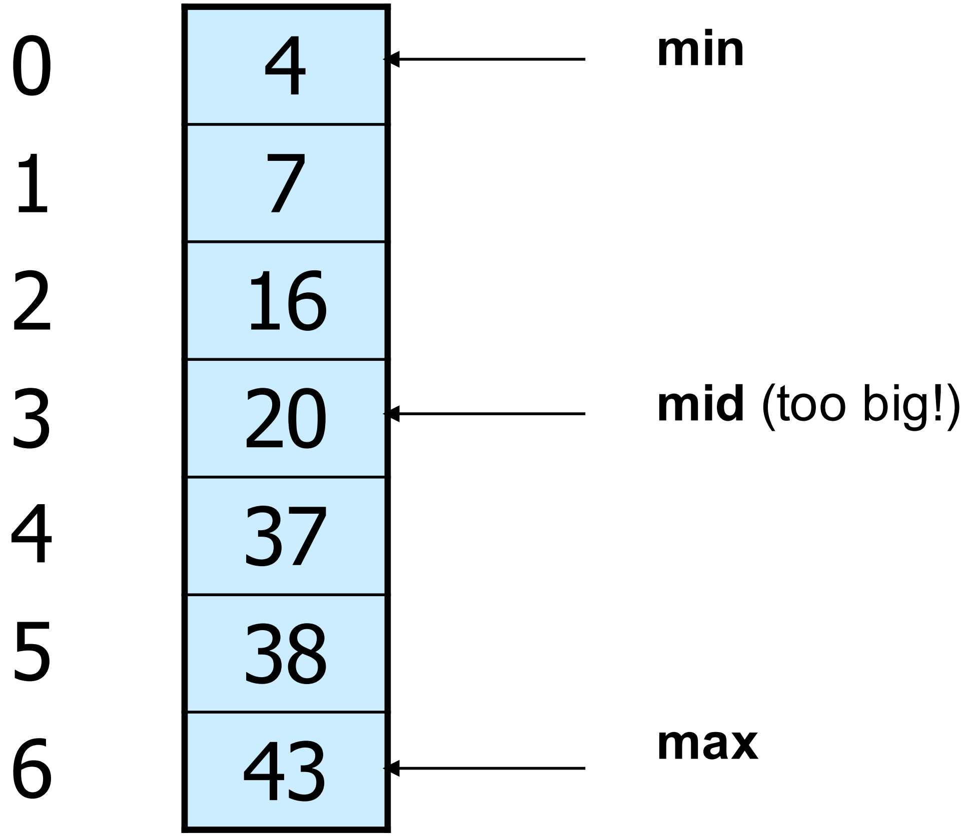
Binary search algorithm

- Algorithm idea: Start in the middle, and only search the portions of the array that might contain the element *i*. Eliminate half of the array from consideration at each step.
  - can be written iteratively, but is harder to get right

- called **binary search** because it chops the area to examine in half each time
  - implemented in Java as method `Arrays.binarySearch` in `java.util` package

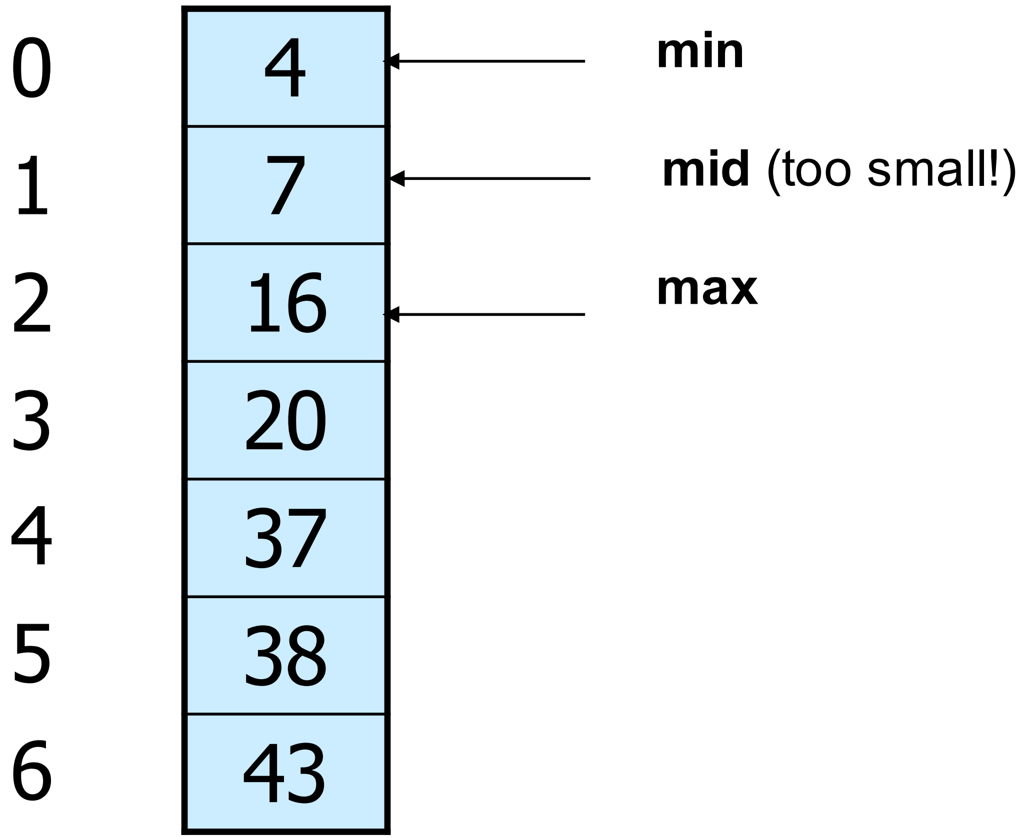
Binary search example

i = 16



Binary search example, cont'd

i = 16



Binary search example, cont'd

i = 16

0	4	
1	7	
2	**16**	**min, mid, max** (found it!)
3	20	
4	37	
5	38	
6	43	

Binary search pseudocode

```
binary search array a for value i:
    if all elements have been searched,
        result is -1.
    examine middle element a[mid].
    if a[mid] equals i,
        result is mid.
    if a[mid] is greater than i,
        binary search left half of a for i.
    if a[mid] is less than i,
        binary search right half of a for i.
```

Runtime of binary search

- How do we analyze the runtime of binary search?
- binary search either exits immediately,
  when input size <= 1 or value found (base case),
  or executes itself on 1/2 as large an input (rec. case)
  - T(1) = c
  - T(2) = T(1) + c
  - T(4) = T(2) + c
  - T(8) = T(4) + c
  - ...
  - T(n) = T(n/2) + c
- How many times does this division in half take place?

Divide-and-conquer algorithms

- **divide-and-conquer algorithm**: a means for solving a problem that first separates the main problem into 2 or more smaller problems, then solves each of the smaller problems, then uses those sub-solutions to solve the original problem
  - 1: "divide" the problem up into pieces
  - 2: "conquer" each smaller piece
  - 3: (if necessary) combine the pieces at the end to produce the overall solution

- binary search is one such algorithm

Recurrences, in brief

- How can we prove the runtime of binary search?

- We can say that the runtime for a given input size *n* is T(*n*).
- At each step of the binary search, we do a constant number *c* of operations, and then we run the same algorithm on 1/2 the original amount of input. Therefore:

  – $T(n) = T(n/2) + c$
  – $T(1) = c$

- Since T is used to define itself, this is called a **recurrence relation**.

Solving recurrences

- **Theorem 7.5** (modified):

A recurrence written in the form

T(n) = a * T(n / b) + f(n)

(where f(n) is a function that is O(n^k) for some power k) has a solution such that

$$T(n) = \begin{cases} O(n^{\log_b a}), & a > b^k \\ O(n^k \log n), & a = b^k \\ O(n^k), & a < b^k \end{cases}$$

- this form of recurrence is very common for divide-and-conquer algorithms

Runtime of binary search

- Binary search is of the correct format:
  $T(n) = a * T(n / b) + f(n)$

  - $T(n) = T(n/2) + c$
  - $T(1) = c$

  - $f(n) = c = O(1) = O(n^0)$ ... therefore $k = 0$
  - $a = 1$, $b = 2$

- $1 = 2^0$, therefore:
  $T(n) = O(n^0 \log n) = \mathbf{O(\log n)}$
- (recurrences not needed for our exams)

Recursive backtracking

- **backtracking**: an elegant technique of searching for a solution to a problem by exploring each possible solution to completion, then "backing out" if it is unsatisfactory
  - often implemented using recursion
  - can yield straightforward solutions for otherwise extremely difficult problems
  - a "depth-first" algorithmic technique (tries each possible solution as deeply as possible before giving up)

Backtracking: escape a maze

- Consider the problem of escaping a maze, from start at 's' to exiting the edge of the board.

```
XXXXXXXXXXXXX       XXXXXXXXXXXXX
X   X  XX   X       X   X..XX   X
  X X X XXX X       ..X X.X XXX X
X X X X X  sX       X.X X.X X...X
X X X     X X  -->  X.X X.....X X
X X X X X X X       X.X X.X X X X
X     X X X X       X.....X X X X
XXXXXXXXXXXXX       XXXXXXXXXXXXX
```

Backtracking: escape a maze

- Maze escaping algorithm, cases of interest:
  - if I am on an open square on the edge of the board, ...
  - if I am on a square I have visited before, ...
  - if I am on an unvisited square and not on the edge, ... (look for a way to escape)
- algorithm works because we exhaust one search for a path out before trying another, and we never try a path more than once

More recursion

- Add the following method to our `MyLinkedList` class:

  ```
  public void reverse()
  ```

  It should reverse the order of the elements. Use recursion to solve this problem. Use only a constant amount of external storage.

  – (Is it easier without the recursion?)

Recursion vs. iteration

- every recursive solution has a corresponding iterative solution
  - For example, N! can be calculated with a loop
- recursion has the overhead of multiple method invocations
- however, for some problems recursive solutions are often more simple and elegant than iterative solutions
- you must be able to determine when recursion is appropriate

Recursion can perform badly

- The *Fibonacci numbers* are a sequence of numbers F_0, F_1, ... F_n such that:
  $F_0 = F_1 = 1$
  $F_i = F_{i-1} + F_{i-2}$ for any $i > 1$

- *Problem*: Write a method `fib` that, when given an integer *i*, computes the i^{th} Fibonacci number.

- Why might a recursive solution to this problem be a bad idea? (Let's write it...)
  - Can we fix it? If so, how?

Revisiting Fibonacci...

- recursive Fibonacci was expensive because it made many, many recursive calls
  - fibonacci(n) recomputed fibonacci(n-1 ... 1) many times in finding its answer!
  - this is a common case of "overlapping subproblems" or "divide poorly and reconquer", where the subtasks handled by the recursion are redundant with each other and get recomputed
  - is there a way that we could optimize this runtime to avoid these unneeded calls?

Dynamic programming

- **dynamic programming**: saving results of subproblems so that they do not need to be recomputed, and can be used in solving other subproblems
  - example: saving results from sub-calls in a list or table
  - can dramatically speed up the number of calls for a recursive function with overlapping subproblems
  - what is the change in Big-Oh for fibonacciR? (non-trivial to calculate exactly)

Dynamic fibonacci pseudocode

```
table[]   {table[0] = 1, table[2] = 1}
max = 2

fibonacciR(n):
  n is in table, return table[n]
  else
    compute fibonacciR(n-1) and fibonacciR(n-2)
    add them to get table[n] = fibonacciR(n)
    set max = n
```

- Let's write it…