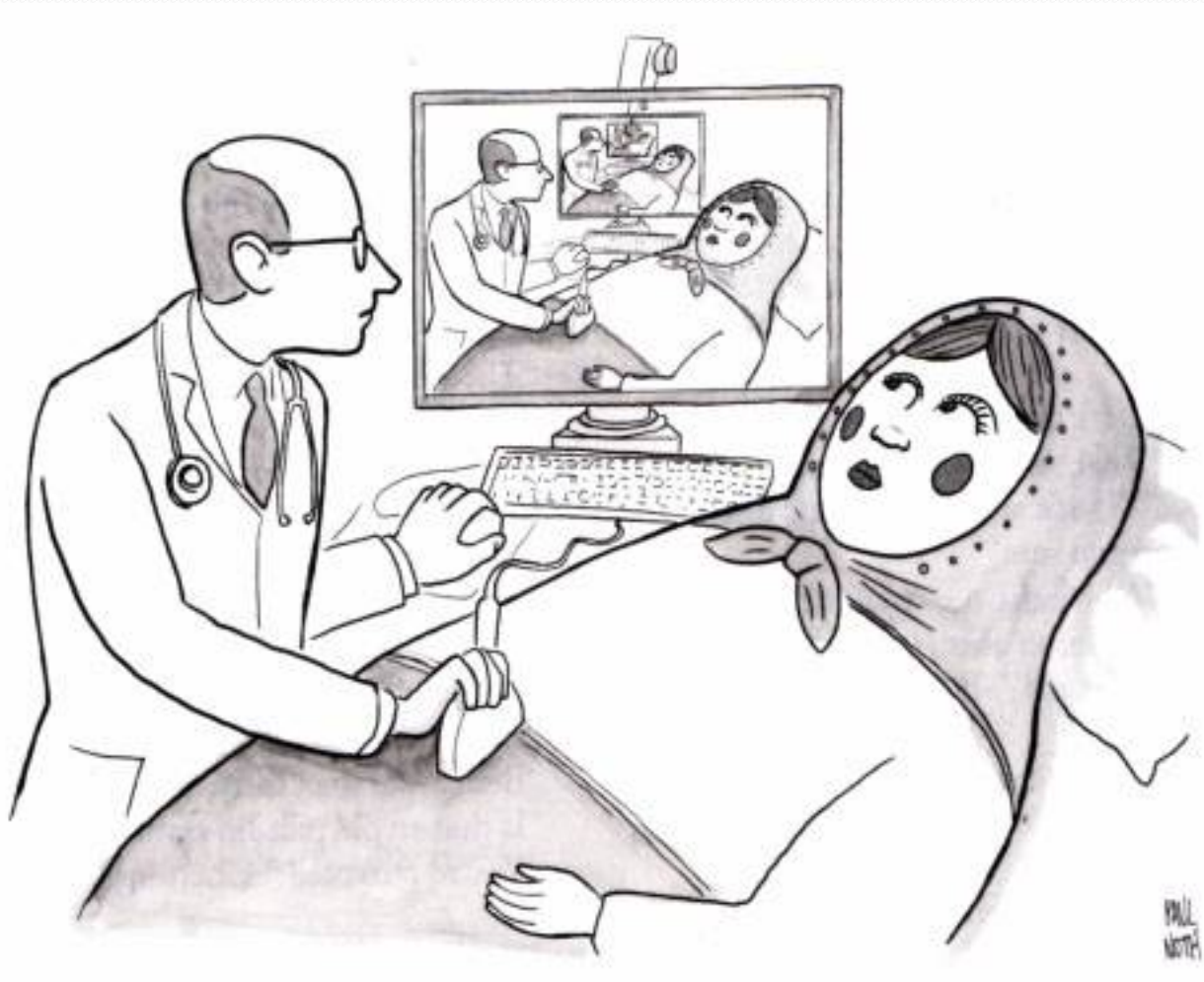


Building Java Programs

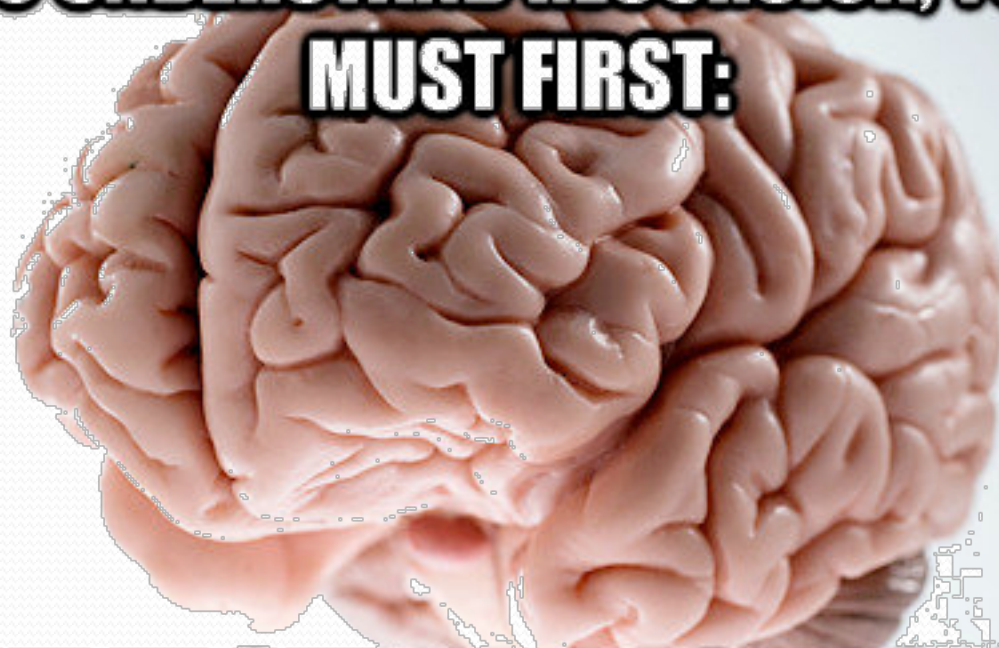
Chapter 12: Recursive public/private pairs

Chapter 13: Searching

reading: 13.3



**TO UNDERSTAND RECURSION, YOU
MUST FIRST:**



UNDERSTAND RECURSION.

quikmemes.com

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
 - **base case**: simple problem that can be solved directly.
 - **recursive case**: more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
- Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
- A crucial part of recursive programming is identifying these cases.

Recursion Challenges

- Forgetting a base case
 - Infinite recursion resulting in `StackOverflowError`
- Working away from the base case
 - The recursive case must make progress towards the base case
 - Infinite recursion resulting in `StackOverflowError`
- Running out of memory
 - Even when making progress to the base case, some inputs may require too many recursive calls: `StackOverflowError`
- Recomputing the same subproblem over and over again
 - Refining the algorithm could save significant time

Exercise

- Write a method `crawl` accepts a `File` parameter and prints information about that file.
 - If the `File` object represents a normal file, just print its name.
 - If the `File` object represents a directory, print its name and information about every file/directory inside it, indented.

```
cse143
  handouts
    syllabus.doc
    lecture_schedule.xls
  homework
    1-tiles
      TileMain.java
      TileManager.java
      index.html
      style.css
```

- **recursive data:** A directory can contain other directories.

File objects

- A `File` object (from the `java.io` package) represents a file or directory on the disk.

Constructor/method	Description
<code>File(String)</code>	creates <code>File</code> object representing file with given name
<code>canRead()</code>	returns whether file is able to be read
<code>delete()</code>	removes file from disk
<code>exists()</code>	whether this file exists on disk
<code>getName()</code>	returns file's name
<code>isDirectory()</code>	returns whether this object represents a directory
<code>length()</code>	returns number of bytes in file
<code>listFiles()</code>	returns a <code>File[]</code> representing files in this directory
<code>renameTo(File)</code>	changes name of file

Public/private pairs

- We cannot vary the indentation without an extra parameter:

```
public static void crawl(File f, String indent) {
```

- Often the parameters we need for our recursion do not match those the client will want to pass.

In these cases, we instead write a pair of methods:

- 1) a public, non-recursive one with parameters the client wants
- 2) a private, recursive one with the parameters we really need

Exercise solution 2

```
// Prints information about this file,  
// and (if it is a directory) any files inside it.  
public static void crawl(File f) {  
    crawl(f, "");    // call private recursive helper  
}  
  
// Recursive helper to implement crawl/indent  
// behavior.  
private static void crawl(File f, String indent) {  
    System.out.println(indent + f.getName());  
    if (f.isDirectory()) {  
        // recursive case; print contained files/dirs  
        for (File subFile : f.listFiles()) {  
            crawl(subFile, indent + "    ");  
        }  
    }  
}
```

Recursive Data

- A file is one of
 - A simple file
 - A directory containing files
- Directories can be nested to an arbitrary depth
- Iterative code to crawl a directory structure requires data structures
 - In recursive solution, we use the call stack

Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
 - Can be implemented with a loop or recursively
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating a binary search on a sorted array. The array is shown with indices 0 to 16 and corresponding values. The value 42 is highlighted in yellow at index 10. Below the array, three boxes labeled 'min', 'mid', and 'max' are shown with arrows pointing to the corresponding indices in the array: 'min' points to index 0, 'mid' points to index 8, and 'max' points to index 16.

Binary search code

```
// Returns the index of an occurrence of target in a,  
// or a negative number if the target is not found.  
// Precondition: elements of a are in sorted order  
public static int binarySearch(int[] a, int target) {  
    int min = 0;  
    int max = a.length - 1;  
  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (a[mid] < target) {  
            min = mid + 1;  
        } else if (a[mid] > target) {  
            max = mid - 1;  
        } else {  
            return mid;    // target found  
        }  
    }  
  
    return -(min + 1);    // target not found  
}
```

Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
 - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index = binarySearch(data, 42); // 10
int index2 = binarySearch(data, 66); // -14
```

Ordering and objects

- Can we compare Strings?
 - Operators like `<` and `>` do not work with `String` objects.
 - But we do think of strings as having an alphabetical ordering.
- **natural ordering**: Rules governing the relative placement of all values of a given type.
- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
 - $A < B$, $A == B$, $A > B$

The compareTo method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
 - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```
- A call of `A.compareTo(B)` will return:
 - a value < 0 if **A** comes "before" **B** in the ordering,
 - a value > 0 if **A** comes "after" **B** in the ordering,
 - or 0 if **A** and **B** are considered "equal" in the ordering.

Using compareTo

- compareTo can be used as a test in an if statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
if (a < b) { ...	if (a.compareTo(b) < 0) { ...
if (a <= b) { ...	if (a.compareTo(b) <= 0) { ...
if (a == b) { ...	if (a.compareTo(b) == 0) { ...
if (a != b) { ...	if (a.compareTo(b) != 0) { ...
if (a >= b) { ...	if (a.compareTo(b) >= 0) { ...
if (a > b) { ...	if (a.compareTo(b) > 0) { ...

Exercise solution

```
// Returns the index of an occurrence of the given value in
// the given array, or a negative number if not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

// Recursive helper to implement search behavior.
private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) {
        return -1;           // target not found
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {           // too small; go right
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) { // too large; go left
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;           // target found; a[mid] == target
        }
    }
}
}
```

Binary search runtime

- For an array of size N , it eliminates $\frac{1}{2}$ until 1 element remains.

$N, N/2, N/4, N/8, \dots, 4, 2, 1$

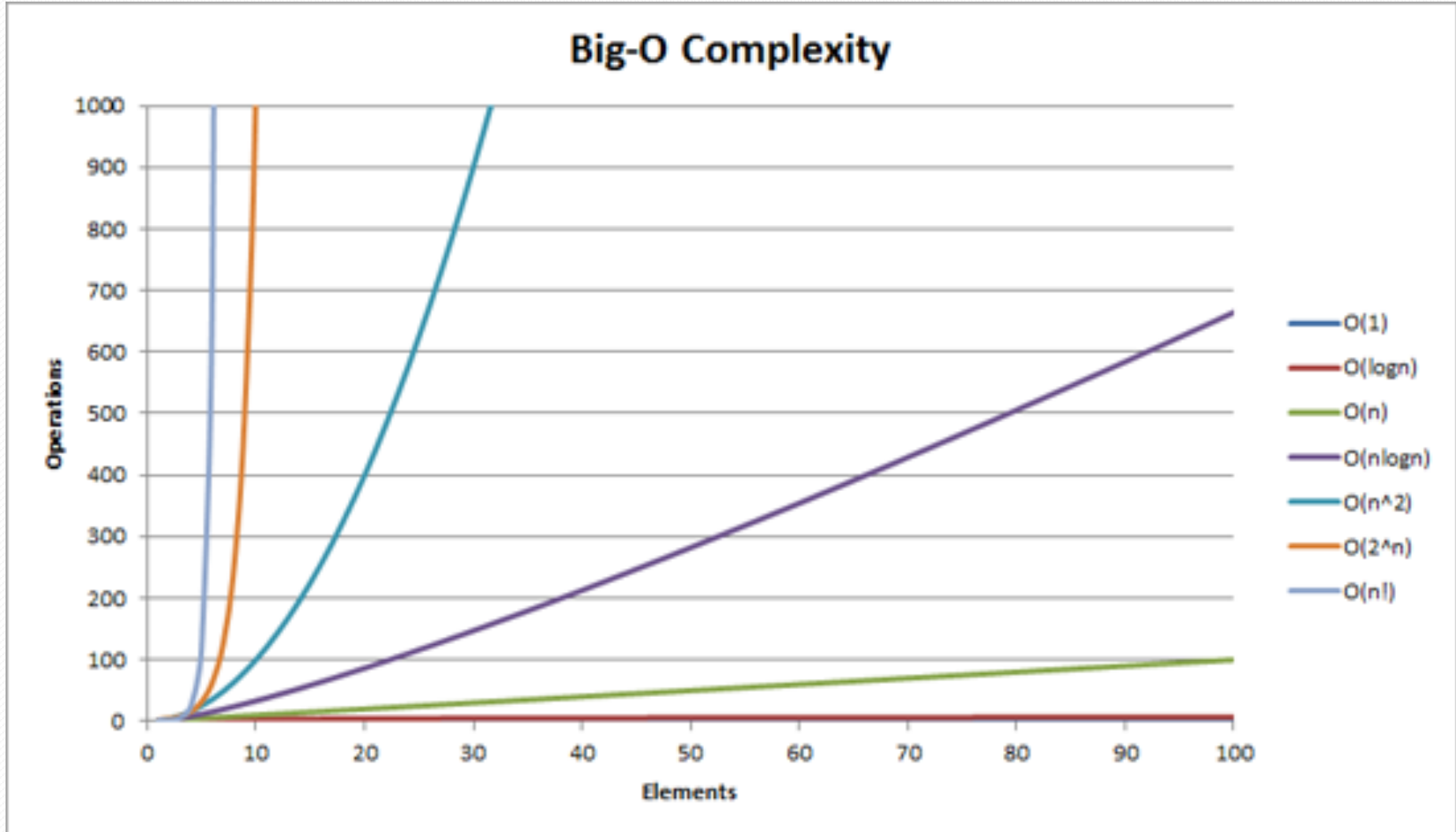
- How many divisions does it take?
- Think of it from the other direction:
 - How many times do I have to multiply by 2 to reach N ?
 $1, 2, 4, 8, \dots, N/4, N/2, N$
 - Call this number of multiplications " x ".

$$2^x = N$$

$$\mathbf{x = \log_2 N}$$

- Binary search is in the **logarithmic** complexity class.

Complexity classes



Recursive Graphics

- See section 12.4

