Divide and Conquer Strategy for Problem Solving - Recursive Functions

Atul Prakash

References:
1. Ch. 4, Downey.
2. Introduction to Recursion Online Notes Ch. 70 by B. Kjell: http://chortle.ccsu.edu/java5/Notes/chap70/ch70_1.html
3.
Divide and Conquer

• Basic Idea of Divide and Conquer:
  • If the problem is easy, solve it directly
  • If the problem cannot be solved as is, decompose it into smaller parts, solve the smaller parts
Some Examples

- Finding the exit from within a hotel
- Finding your car in a parking lot
- Looking up a name in a phone book
Mental Exercise

• You are at the end of a very long hotel lobby with a long series of doors, with one door next to you. You are looking for the door that leads to the fire exit.

• What is your first step?
First Step
What do you do if the first step does not work?
FindExit Strategy

- Try the door next to you for the exit
- If it does not lead to an exit, advance to the next door. And repeat the FindExit strategy
Elements of the Solution

• Try a direct solution: check the nearby door for the exit

• If it does not work, use the same strategy on the smaller problem after advancing to the next door
Recursion in Words of Wisdom

- Philosopher Lao-tzu:
  - The journey of a thousand miles begins with a single step
Break a Stone into Dust

- BreakStone: You want to ground a stone into dust (very small stones)
- What is your first step?
First Step

- Use a hammer and strike the stone
Next Step

• If a stone pieces that result is small enough, we are done with that part

• For pieces that are too large, repeat the BreakStone process
Common Elements

• If the problem is small enough to be solved directly, do it

• If not, find a smaller problem and use its solution to create the solution to the larger problem
Looking up a Phone Number

• You have a phone book with names in alphabetical order

• Given a name, what is your first step?
First Step

• Open the phone book in the middle (or on a random page)

• If name within the range of names on that page, find it, and we are done
Smaller Problem Step

• If name not in the page, you can exclude either the left part or the right part

• Search in the remaining part
After another step
Other Problems

- Recursively-defined functions
- Factorial: n!
- Fibonacci numbers
- Ackermann Function
- Tower of Hanoi
- Fractals
- Tree and data searches
Glue in Divide and Conquer

- Often, the parts must be “glued” into a solution

```
Large problem
```

```
Partition 1
Partition 2
Partition 3
Partition 4
```

```
Decompose
```

```
Glue → Solution
```

Factorial Problem

• Example: Finding factorial of \( n \geq 1 \)

• \( n! = n(n-1)(n-2)\ldots 1 \)

• Divide and Conquer Strategy:
  
  • if \( n = 1 \): \( n! = 1 \) (direct solution),
  
  else: \( n! = n \times (n-1)! \)

\[
\text{factorial}(n) = n \times \text{factorial}(n-1)
\]

\[
\text{factorial}(1) = 1
\]
Divide and Conquer

factorial(n) \rightarrow factorial(n-1) \rightarrow n \rightarrow * \rightarrow n!
Recursion in Functions

- When a function makes use of itself, as in a divide-and-conquer strategy, it is called recursion.

- Recursion requires:
  - Base case or direct solution step. (e.g., factorial(1))

- Recursive step(s):
  - A function calling itself on a smaller problem. E.g., n*factorial(n-1)

- Eventually, all recursive steps must reduce to the base case.
• Definition: \( n! \) is defined as 1 if \( n = 0 \) (direct solution),

Otherwise, \( n! = n \times (n-1)! \) (divide-and-conquer)

```java
public static int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
    // post-condition: returns n!
}
```
Stack Diagram

factorial(3)
(local n is 3)

factorial(n = 2)
(local n is 2)

factorial(1)
(local n is 1)

factorial(0)
(local n is 0)

6

2

1

1

1
Understanding Recursive Programs

• Why does it work?

```java
public static int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
    // post-condition: returns n!
}
```

Proof by induction:
(1) Solution works for \( n = 0 \)
(2) If it works for \( n-1 \), it works for \( n \)
(3) 1. and 2. imply, it works for \( n = 1 \)
(4) 2. and 3. imply it works for \( n = 2 \) and in fact any larger \( n \)
Handling Errors

• What if $n$ is $< 1$ in the factorial program?
  • $\text{factorial}(-1)$ will call $\text{factorial}(-2)$, which will call $\text{factorial}(-3)$, etc.
  • Recursion will never reach the base case
  • Document pre-conditions and post-conditions

```
public static int factorial(int n) {
    assert(n >= 0); // pre-condition
    if (n == 0) return 1;
    else return n * factorial(n-1);
    // post-condition: returns n!
}
```
Tower of Hanoi

- Initial state: n disks in decreasing order of size on one peg
- Goal: move all the disks to the 2nd peg.
- Move one disk at a time
- Constraint: a disk can never be on top of a smaller disk

Source: wikipedia. Copied under Wikimedia Common License
Divide-and-Conquer Strategy

• If n is 1, the solution is trivial. Just move the disk to the desired peg

• For n > 1, let's assume we know how to solve the problem for n-1 disks.

• Can we use that to construct a solution for n disks?
Solution Strategy

- Base case: if $n$ is 1, the solution is trivial. Just move the disk.

- Otherwise:
  - Move $(n-1)$ disks from peg A to peg C using Hanoi for $(n-1)$ disks.
  - Move the left-over disk from page A to peg B.
  - Move $(n-1)$ disks from peg C to peg B using Hanoi for $(n-1)$ disks.
public static void move(Object pegA, Object pegB) {
    System.out.println("move disk from " + pegA + " to " + pegB);
}

public static void hanoi(int n, Object pegA, Object pegB, Object pegC) {
    // precondition: n >= 1. disks are on pegA
    assert(n >= 1);
    if (n == 1) {
        move(pegA, pegB);
    } else {
        hanoi(n-1, pegA, pegC, pegB);
        move(pegA, pegB);
        hanoi(n-1, pegC, pegB, pegA);
    }
    // post-condition: top n disks moved from pegA to pegB
}

hanoi(3, "peg 1", "peg 2", "peg 3");

Show a run in Eclipse
Tower of Hanoi Analysis

```python
>>> recursion.hanoi(2, 'peg 1', 'peg 2', 'peg 3')
move disk from peg 1 to peg 3
move disk from peg 1 to peg 2
move disk from peg 3 to peg 2

>>> recursion.hanoi(2, 'peg 1', 'peg 3', 'peg 2')
move disk from peg 1 to peg 2
move disk from peg 1 to peg 3
move disk from peg 2 to peg 3

>>> recursion.hanoi(2, 'peg 3', 'peg 2', 'peg 1')
move disk from peg 3 to peg 1
move disk from peg 3 to peg 2
move disk from peg 1 to peg 2

>>> recursion.hanoi(3, 'peg 1', 'peg 2', 'peg 3')
move disk from peg 1 to peg 2
move disk from peg 1 to peg 3
move disk from peg 2 to peg 3

move disk from peg 1 to peg 2
move disk from peg 3 to peg 1
move disk from peg 3 to peg 2
move disk from peg 1 to peg 2
```

Tower of Hanoi is pretty slow for larger values of n.

Q: How many disk moves (approximately) for a given n?

Note: The above is on Python version. Java is analogous.
Performance

TOWER OF HANOI really slows down for large n

Factorial's time is roughly linear with n.
We say that the time for factorial is $O(n)$, called Big-Oh(n), or linear in n
Big-Oh

- A way to measure how execution time or memory use will grow with input size

- Formally, $f(n)$ is $O(g(n))$ iff for sufficiently large values of $n$, $f(n)$ is within constant times of $g(n)$. That is,

- $f(n) < c \cdot g(n)$ for all $n > N$ and some constant $c$. 
Big-Oh examples

- $3n + 2$ is $O(n)$ because $3n+1 < 4n$ for large $n$
- $1000n + 100000$ is also $O(n)$
- $10n^2 + 3$ is $O(n^2)$
- $2^n + n^3$ is $O(2^n)$
Basic Points

• Ignore the small stuff
  • n + 10: ignore the 10
  • n^2 + n: ignore the n

• Simplify
  • Replace 10 by 1. Both are O(1)
  • 2n can be replaced by n. Both are O(n)
Factorial Time Analysis

- Factorial of 0: constant time.
  - $T(0) = 1$ (treat constants as 1)
- Time required to compute factorial of $n$:
  - $T(n) = T(n-1) + 1$ (treat constants as 1)
    
    
    $T(4) = T(3) + 1$
    
    $= T(2) + 1 + 1$
    
    $= T(1) + 1 + 1 + 1$
    
    $= T(0) + 1 + 1 + 1 + 1 = 5$

In general, $T(n)$ is $n+1$ or $O(n)$
Hanoi: Number of Moves

• Let $T(n) =$ number of disk moves for $n$ disks
• $T(1) = 1$
• $T(2) = 2 \times T(1) + 1 = 3$
• $T(3) = 2 \times T(2) + 1 = 7$
• $T(4) = 2 \times T(3) + 1 = 15$
• See a pattern?

• $T(n) = 2^n - 1$ or $O(2^n)$
• Hanoi for 64 disks would take a very, very long time!
• This is an example of an exponential-time program.
Advantage of Big-Oh Analysis

- Big-Oh gives you trends versus problem size
- Big-Oh analysis holds even if computers become 10 times faster
Common Growth Rates

- $O(1)$: constant time. For example, array lookup, given an index
- $O(n)$: linear time. For example, scan an array of length $n$ for a value
- $O(\log n)$: Between constant and linear time.
- $O(2^n)$: Exponential time. Very bad

We will see lots of examples later
Fractals

- Fractals are recursive drawings. They occur a lot in nature and there is a field called fractal geometry. Can use recursion to draw them.

But, how to do drawings in Java?
Drawing in Java

- Java has several graphics packages: awt, swing, etc.
- We will use ACM Graphics package for Java, as it is designed for educational use
- Download acm.jar and*.java from CTools in 11-lecture-code folder
- Documentation:
  - [http://jtf.acm.org/tutorial/Introduction.html](http://jtf.acm.org/tutorial/Introduction.html)
Using acm.jar

Command line (use semi-colons on Windows):

```
javac -cp .:/path/to/acm.jar *.java
java -cp .:/path/to/acm.jar MainClass
```

- In Eclipse, go to Project -> Properties -> Java Build Path
- Add acm.jar to the build path
ACM Graphics Package

• Create shapes, e.g., GLabel, GLine, GTurtle, etc. in a GraphicsProgram
• Add them to the canvas using the add() routine
• getWidth() and getHeight() return the height of a canvas.
• Coordinate system: Top-left corner is (0,0).
Mini-exercise

• Compile and run one of the Hello programs that use the ACM jar file. Submit a screen snapshot

• Generate the stack of squares and submit a screen snapshot
public static void drawSquare(GTurtle t, double len) {
    t.penDown();
    for (int i = 0; i < 4; i++) {
        t.forward(len);
        t.left(90);
    }
}

public void run() {
    // Place turtle in the center of the canvas
    GTurtle turtle = new GTurtle(getWidth()/2.0, getHeight()/2.0);
    add(turtle);
    drawSquare(turtle, 100.0);
}
public class HelloGraphics extends GraphicsProgram {

   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("SansSerif-100");
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() + label.getAscent()) / 2;
      add(label, x, y);
   }

   /* Standard Java entry point. Call MainClass.start(args)
    to get graphics program going */

   public static void main(String[] args) {
      new HelloGraphics().start(args);
   }
}
public void drawStack(GTurtle t, double len, int squarecount) {
    // precondition: turtle at "origin"
    if (squarecount == 0) return;
    drawSquare(t, len);    // draw big square, ending at the start location.
    t.left(90); t.forward(len); t.right(90);    // go to the top-left
    drawStack(t, len/2.0, squarecount-1);
    t.right(90); t.forward(len); t.left(90);    // return to origin
    // post-condition: draw the stack and return to origin
}

public void run() {
    GTurtle turtle = new GTurtle(getWidth()/2.0, getHeight()/2.0);
    add(turtle);
    drawStack(turtle, 100.0, 3);
}
Another Way

• Use shape drawing functions rather than turtles. Basic primitive
  • draw a shape of a given size at \((x, y)\)
  • Shapes include lines, squares, circles, rectangles, etc.
  • Shapes can have attributes, such as line thickness, color, fill, etc.
Drawing a square

// File: SquareStackWithShapes.java

// draws a square at (x, y) of length len. Origin top-left corner.
public void drawSquare(double x, double y, double len) {
    GRect r = new GRect(x, y, len, len);
    add(r);
}

public void run() {
    drawSquare(getWidth()/2, getHeight()/2, 100.0);
}
Drawing a Stack

// draw a stack squarecount deep at (x, y), with squares becoming
// half the size as you go up the stack.
public void drawStack(double x, double y, double len, int squarecount) {
    if (squarecount == 0) return;
    drawSquare(x, y, len);  // draw big square, ending at the start location.
    drawStack(x, y-len/2.0, len/2.0, squarecount-1);
}

Draw the big square. Note: origin top-left corner.
Draw the remaining stack with origin 1/2 length up.
public void drawTreeOfSquares(double x, double y, double len, int squarecount) {
    if (squarecount == 0) return;
    drawSquare(x, y, len);
    drawTreeOfSquares(x-len*0.25, y-len*0.5, len*0.5, squarecount-1);
    drawTreeOfSquares(x+len*0.75, y-len*0.5, len*0.5, squarecount-1);
}
Hierarchical Data (trees)

Example: child-dad/mom relationship
Count nodes in a Tree

If tree empty, return 0
Else, return
one
+ count of left subtree
+ count of right subtree
Summary

• Divide and Conquer is a common problem solving strategy
• It often maps to recursive algorithms
• Big-Oh notation a way to estimate how time required to solve a problem will grow as the problem size increases