6. Recursion

Overview

Q. What is recursion?
A. When something is specified in terms of itself.

Why learn recursion?
• Represents a new mode of thinking.
• Provides a powerful programming paradigm.
• Enables reasoning about correctness.
• Gives insight into the nature of computation.

Many computational artifacts are naturally self-referential.
• File system with folders containing folders.
• Fractal graphical patterns.
• Divide-and-conquer algorithms (stay tuned).

Example: Convert an integer to binary

Recursive program
To compute a function of a positive integer \( N \)
• Base case. Return a value for small \( N \).
• Reduction step. Assuming that it works for smaller values of its argument, use the function to compute a return value for \( N \).

```java
public class Binary {
    public static String convert(int N) {
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        StdOut.println(convert(N));
    }
}
```

Q. How can we be convinced that this method is correct?
A. Use mathematical induction.

Foundations
• A classic example
• Recursive graphics
• Avoiding exponential waste
• Dynamic programming

Section 2.3
Mathematical induction (quick review)

**To prove a statement involving a positive integer** $N$

- **Base case.** Prove it for some specific values of $N$.
- **Induction step.** Assuming that the statement is true for all positive integers less than $N$, use that fact to prove it for $N$.

**Example**

The sum of the first $N$ odd integers is $N^2$.

**Base case.** True for $N = 1$.

**Induction step.** The $N$th odd integer is $2N - 1$. Let $T_N = 1 + 3 + 5 + \ldots + (2N - 1)$ be the sum of the first $N$ odd integers.
- Assume that $T_{N-1} = (N - 1)^2$.
- Then $T_N = (N - 1)^2 + (2N - 1) = N^2$.

An alternate proof

---

Proving a recursive program correct

**Recursion**

To compute a function of $N$

- **Base case.** Return a value for small $N$.
- **Reduction step.** Assuming that it works for smaller values of its argument, use the function to compute a return value for $N$.

**Mathematical induction**

To prove a statement involving $N$

- **Base case.** Prove it for small $N$.
- **Induction step.** Assuming that the statement is true for all positive integers less than $N$, use that fact to prove it for $N$.

**Recursive program**

```java
public static String convert(int N) {
    if (N == 1) return "1";
    return convert(N/2) + (N % 2);  
}
```

**Correctness proof, by induction**

`convert()` computes the binary representation of $N$

- **Base case.** Returns "1" for $N = 1$.
- **Induction step.** Assume that `convert()` works for $N/2$
  1. Correct to append "0" if $N$ is even, since $N = 2(N/2)$.
  2. Correct to append "1" if $N$ is odd since $N = 2(N/2) + 1$.

---

Mechanics of a function call

System actions when any function is called

- Save environment (values of all variables and call location).
- Initialize values of argument variables.
- Transfer control to the function.
- Restore environment (and assign return value)
- Transfer control back to the calling code.

```java
public class Binary {
    public static String convert(int N) {
        if (N == 1) return "1";
        return convert(N/2) + (N % 2);  
    } 
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        System.out.println(convert(N));
    }
}
```

**Programming with recursion: typical bugs**

**Missing base case**

```java
public static double bad(int N) {
    return bad(N-1) + 1.0/N;
}
```

**No convergence guarantee**

```java
public static double bad(int N) {
    if (N == 1) return 1.0;
    return bad(1 + N/2) + 1.0/N;
}
```

Both lead to infinite recursive loops (bad news).
Collatz Sequence

Collatz function of $N$.
- If $N$ is 1, stop.
- If $N$ is even, divide by 2.
- If $N$ is odd, multiply by 3 and add 1.

```
public static void collatz(int N) {
    System.out.println(N + " ");
    if (N == 1) return;
    if (N % 2 == 0) collatz(N / 2);
    collatz(3*N + 1);
}
```

Amazing fact. No one knows whether or not this function terminates for all $N$.

Note. We usually ensure termination by only making recursive calls for smaller $N$.
Warmup: subdivisions of a ruler (revisited)

ruler(n): create subdivisions of a ruler to 1/2^n inches.
• Return one space for n = 0.
• Otherwise, sandwich n between two copies of ruler(n-1).

Tracing a recursive program

Use a recursive call tree
• One node for each recursive call.
• Label node with return value after children are labeled.

Towers of Hanoi puzzle

A legend of uncertain origin
• n = 64 discs of differing size; 3 posts; discs on one of the posts from largest to smallest.
• An ancient prophecy has commanded monks to move the discs to another post.
• When the task is completed, the world will end.

Rules
• Move discs one at a time.
• Never put a larger disc on a smaller disc.

Q. Generate list of instruction for monks?
Q. When might the world end?

Towers of Hanoi

For simple instructions, use cyclic wraparound
• Move right means 1 to 2, 2 to 3, or 3 to 1.
• Move left means 1 to 3, 3 to 2, or 2 to 1.

A recursive solution
• Move n − 1 discs to the left (recursively).
• Move largest disc to the right.
• Move n − 1 discs to the left (recursively).
Towers of Hanoi solution \( n = 3 \)

Recursive call tree for towers of Hanoi

Structure is the same as for the ruler function and suggests 3 useful and easy-to-prove facts.

- Each disc always moves in the same direction.
- Moving smaller disc always alternates with a unique legal move.
- Moving \( n \) discs requires \( 2^n - 1 \) moves.

Answers for towers of Hanoi

Q. Generate list of instructions for monks?

A. (Long form). 1L 2R 1L 3L 1L 2R 1L 4R 1L 2R 1L 3L 1L 2R 1L 5L 1L 2R 1L 3L 1L 2R 1L 4R ...

A. (Short form). Alternate "1L" with the only legal move not involving the disc 1. "L" or "R" depends on whether \( n \) is odd or even.

Q. When might the world end?

A. Not soon: need \( 2^{2^4} - 1 \) moves.

Note: Recursive solution has been proven optimal.
6: Recursion

- Foundations
- A classic example
- Recursive graphics
- Avoiding exponential waste
- Dynamic programming

Recursive graphics in the wild

"Hello, World" of recursive graphics: H-trees

H-tree of order $n$
- If $n$ is 0, do nothing.
- Draw an H, centered.
- Draw four H-trees of order $n-1$ and half the size, centered at the tips of the H.
**H-trees**

**Application.** Connect a large set of regularly spaced sites to a single source.

---

**Recursive H-tree implementation**

```java
public class Htree
{
    public static void draw(int n, double sz, double x, double y)
    {
        if (n == 0) return;
        double x0 = x - sz/2, x1 = x + sz/2;
        double y0 = y - sz/2, y1 = y + sz/2;
        StdDraw.line(x0, y, x1, y);
        StdDraw.line(x0, y0, x0, y1);
        StdDraw.line(x1, y0, x1, y1);
        StdDraw.line(x0, y1, x1, y0);
        draw(n-1, sz/2, x0, y0);
        draw(n-1, sz/2, x0, y1);
        draw(n-1, sz/2, x1, y0);
        draw(n-1, sz/2, x1, y1);
    }
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        draw(n, .5, .5, .5);
    }
}
```

**Deluxe H-tree implementation**

```java
public class HtreeDeluxe
{
    public static void draw(int n, double sz, double x, double y)
    {
        if (n == 0) return;
        double x0 = x - sz/2, x1 = x + sz/2;
        double y0 = y - sz/2, y1 = y + sz/2;
        StdDraw.line(x0, y, x1, y);
        StdDraw.line(x0, y0, x0, y1);
        StdDraw.line(x1, y0, x1, y1);
        StdDraw.line(x0, y1, x1, y0);
        draw(n-1, sz/2, x0, y0);
        draw(n-1, sz/2, x0, y1);
        draw(n-1, sz/2, x1, y0);
        draw(n-1, sz/2, x1, y1);
    }
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        draw(n, .5, .5, .5);
    }
}
```

---

**Fractional Brownian motion**

A process that models many phenomenon.
- Price of stocks.
- Dispersion of fluids.
- Rugged shapes of mountains and clouds.
- Shape of nerve membranes.

---

**Brownian bridge model**

An actual mountain

**Price of an actual stock**

Black-Scholes model (two different parameters)
Fractional Brownian motion simulation

**Midpoint displacement method**
- Consider a line segment from \((x_0, y_0)\) to \((x_1, y_1)\).
- If sufficiently short draw it and return
- Divide the line segment in half, at \((x_m, y_m)\).
- Choose \(\Delta\) at random from Gaussian distribution.
- Add \(\Delta\) to \(y_m\).
- Recur on the left and right line segments.

\[
\begin{align*}
(x_0, y_0) & \quad \text{\textbullet} \\
(x_m, y_m) & \quad \text{\textbullet} \\
(x_1, y_1) & \quad \text{\textbullet}
\end{align*}
\]

Brownian motion implementation

```java
public class Brownian {
    public static void curve(double x0, double y0, double x1, double y1, double var, double s) {
        if (x1 - x0 < .01) {
            StdDraw.line(x0, y0, x1, y1); return; }
        double x_m = (x0 + x1) / 2;
        double y_m = (y0 + y1) / 2;
        double stddev = Math.sqrt(var);
        double delta = StdRandom.gaussian(0, stddev);
        curve(x0, y0, x_m + delta, y_m, var/s, s);
        curve(x_m, y_m + delta, x1, y1, var/s, s);
    }
    public static void main(String[] args) {
        double hurst = Double.parseDouble(args[0]);
        double s = Math.pow(2, 2*hurst);
        curve(0, .5, 1.0, .5, .01, s); // control parameter
    }
}
```

A 2D Brownian model: plasma clouds

**Midpoint displacement method**
- Consider a rectangle centered at \((x, y)\) with pixels at the four corners.
- If the rectangle is small, do nothing.
- Color the midpoints of each side the average of the endpoint colors.
- Choose \(\Delta\) at random from Gaussian distribution.
- Color the center pixel the average of the four corner colors plus \(\Delta\).
- Recurse on the four quadrants.

Booksight code actually draws a rectangle to avoid artifacts.

A Brownian cloud
6. Recursion

- Foundations
- A classic example
- Recursive graphics
- Avoiding exponential waste
- Dynamic programming

Fibonacci numbers

Let $F_n = F_{n-1} + F_{n-2}$ for $n > 1$ with $F_0 = 0$ and $F_1 = 1$.

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | ...
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|-----|
| $F_n$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | ...

Models many natural phenomena and is widely found in art and architecture.

Examples.
- Model for reproducing rabbits.
- Nautilus shell.
- Mona Lisa.
- ...

Facts (known for centuries).
- $F_n / F_{n-1} \Rightarrow \Phi = 1.618...$ as $n \to \infty$.
- $F_n$ is the closest integer to $\Phi^{n+1}/\sqrt{5}$. 

Leonardo Fibonacci
C. 1170 – C. 1250

golden ratio $F_n / F_{n-1}$

13 21

21 8
Fibonacci numbers and the golden ratio in the wild

Computing Fibonacci numbers

Q. [Curious individual.] What is the exact value of $F_{60}$?

A. [Novice programmer.] Just a second. I’ll write a recursive program to compute it.

```
public class FibonacciR {
    public static long F(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        return F(n-1) + F(n-2);
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(F(n));
    }
}
```

Exponential waste

Let $C_n$ be the number of times $F(n)$ is called when computing $F(60)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$C_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>$F_1$</td>
</tr>
<tr>
<td>59</td>
<td>$F_2$</td>
</tr>
<tr>
<td>58</td>
<td>$F_3$</td>
</tr>
<tr>
<td>57</td>
<td>$F_4$</td>
</tr>
<tr>
<td>56</td>
<td>$F_5$</td>
</tr>
<tr>
<td>55</td>
<td>$F_6$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>$&gt;2.5\times10^{12}$ $F_{61}$</td>
</tr>
</tbody>
</table>

Exponentially wasteful to recompute all these values.
Exponential waste dwarfs progress in technology

If you engage in exponential waste, you will not be able to solve a large problem.

<table>
<thead>
<tr>
<th>1970s</th>
<th>2010s: 10,000+ times faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>time to compute $F_n$</td>
</tr>
<tr>
<td>30</td>
<td>minutes</td>
</tr>
<tr>
<td>40</td>
<td>hours</td>
</tr>
<tr>
<td>50</td>
<td>weeks</td>
</tr>
<tr>
<td>60</td>
<td>years</td>
</tr>
<tr>
<td>70</td>
<td>centuries</td>
</tr>
<tr>
<td>80</td>
<td>millenia</td>
</tr>
</tbody>
</table>

1970s: "That program won't compute $F_{30}$ before you graduate!"

2010s: "That program won't compute $F_{80}$ before you graduate!"

Avoiding exponential waste

Memoization

- Maintain an array memo[] to remember all computed values.
- If value known, just return it.
- Otherwise, compute it, remember it, and then return it.

```java
public class FibonacciM {
    static long[] memo = new long[100];
    public static long F(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        if (memo[n] != 0) memo[n] = F(n-1) + F(n-2);
        return memo[n];
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(F(n));
    }
}
```

Simple example of dynamic programming (next).
An alternative to recursion that avoids recomputation

Dynamic programming.
- Build computation from the "bottom up".
- Solve small subproblems and save solutions.
- Use those solutions to build bigger solutions.

Fibonacci numbers

```java
class Fibonacci {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        long[] F = new long[n+1];
        F[0] = 0; F[1] = 1;
        for (int i = 2; i <= n; i++)
            F[i] = F[i-1] + F[i-2];
        System.out.println(F[n]);
    }
}
```

Key advantage over recursive solution. Each subproblem is addressed only once.

DP example: Longest common subsequence

Def. A subsequence of a string s is any string formed by deleting characters from s.

Ex 1. s = ggcaccagc
    = gca
    = ggcaccac
    = ggcaccacg
    = ggcaccacgc
    = ggcaccacgcg
    ...

Ex 2. t = acgcgcgatacg
    = acgcgcgcg
ggcg
    = acgcgcgatacg
    = acgcgcgatacg
    = acgcgcgatacg
    ...

Def. The LCS of s and t is the longest string that is a subsequence of both.

Goal. Efficient algorithm to compute the LCS and/or its length

Longest common subsequence

Goal. Efficient algorithm to compute the length of the LCS of two strings s and t.

Approach. Keep track of the length of the LCS of s[i..M] and t[j..N] in opt[i, j]

Three cases:
- i = M or j = N
  opt[i][j] = 0
- s[i] = t[j]
  opt[i][j] = opt[i+1, j+1] + 1
- otherwise
  opt[i][j] = max(opt[i, j+1], opt[i+1][j])

Ex: i = 6, j = 7
s[6..9] = acg
t[6..12] = atacgcg
LCS(acg, tacgcg) = cg
LCS(0, atacgcg) = acg

Ex: i = 6, j = 4
s[6..9] = acg
t[4..12] = cgatacg
LCS(acg, cgatacg) = acg
LCS(acg, cgacg) = cg
LCS(0, cgatacg) = acg

LCS example
LCS length implementation

```java
public class LCS {
    public static void main(String[] args) {
        String s = args[0];
        String t = args[1];
        int M = s.length();
        int N = t.length();
        int[][] opt = new int[M+1][N+1];
        for (int i = 1; i <= M; i++)
            for (int j = 1; j <= N; j++)
                if (s.charAt(i-1) == t.charAt(j-1))
                    opt[i][j] = opt[i-1][j-1] + 1;
                else
                    opt[i][j] = Math.max(opt[i-1][j], opt[i][j-1]);
        System.out.println(opt[M][N]);
    }
}
```

Exercise. Add code to print LCS itself (see LCS.java on booksite for solution).

Dynamic programming and recursion

Broadly useful approaches to solving problems by combining solutions to smaller subproblems.

Why learn DP and recursion?

- Represent a new mode of thinking.
- Provide powerful programming paradigms.
- Give insight into the nature of computation.
- Successfully used for decades.

<table>
<thead>
<tr>
<th>recursion</th>
<th>dynamic programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>advantages</td>
<td>Decomposition often obvious. Easy to reason about correctness.</td>
</tr>
<tr>
<td>pitfalls</td>
<td>Potential for exponential waste. Decomposition may not be simple.</td>
</tr>
</tbody>
</table>