Recursion

Chapter 11

Objectives

• become familiar with the idea of recursion
• learn to use recursion as a programming tool
• become familiar with the binary search algorithm as an example of recursion
• become familiar with the merge sort algorithm as an example of recursion

How do you look up a name in the phone book?

One Possible Way

Search:
  middle page = (first page + last page)/2
  Go to middle page;
  If (name is on middle page)
    done; //this is the base case
  else if (name is alphabetically before middle page)
    last page = middle page //redefine search area to front half
    Search //same process on reduced number of pages
  else  //name must be after middle page
    first page = middle page //redefine search area to back half
    Search //same process on reduced number of pages

Recursion: a definition in terms of itself.

Recursion in algorithms:
• Natural approach to some (not all) problems
• A recursive algorithm uses itself to solve one or more smaller identical problems

Recursion in Java:
• Recursive methods implement recursive algorithms
• A recursive method includes a call to itself

Recursive Methods

Must Eventually Terminate

A recursive method must have at least one base, or stopping, case.

• A base case does not execute a recursive call – stops the recursion
• Each successive call to itself must be a “smaller version of itself” – an argument that describes a smaller problem – a base case is eventually reached
Key Components of a Recursive Algorithm Design

1. What is a smaller identical problem(s)?
   - Decomposition

2. How are the answers to smaller problems combined to form the answer to the larger problem?
   - Composition

3. Which is the smallest problem that can be solved easily (without further decomposition)?
   - Base/stopping case

Examples in Recursion

- Usually quite confusing the first time
- Start with some simple examples
  - recursive algorithms might not be best
  - Later with inherently recursive algorithms
  - harder to implement otherwise

Factorial \((N!)\)

- \(M = (N-1)! \times N\) [for \(N > 1\)]
- \(1! = 1\)
- \(3! = 2! \times 3 = (1! \times 2) \times 3 = 1 \times 2 \times 3\)
- Recursive design:
  - Decomposition: \((N-1)!\)
  - Composition: \(\times N\)
  - Base case: \(1!\)

Factorial Method

```java
public static int factorial(int n) {
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    else // base case
        fact = 1;
    return fact;
}
```

```java
public static int factorial(int 3) {
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```java
public static int factorial(int 2) {
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```
```java
public static int factorial(int n)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
```
public static int factorial(int n) {
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}

Execution Trace
(decomposition)
public static int factorial(int n) {
    int fact;
    if (n > 1)  // recursive case (decomposition)
        fact = factorial(n - 1) * n;  // composition
    else // base case
        fact = 1;
    return fact;
}

factorial(4) -> 24

public static int factorial(int n) {
    int fact = 1;  // base case value
    if (n > 1)  // recursive case (decomposition)
        fact = factorial(n - 1) * n;  // composition
    return fact;
}

Fibonacci Numbers

• The $N$th Fibonacci number is the sum of the previous two Fibonacci numbers
• 0, 1, 1, 2, 3, 5, 8, 13, ...
• Recursive Design:
  – Decomposition & Composition
    • fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  – Base case:
    • fibonacci(1) = 0
    • fibonacci(2) = 1

fibonacci(1) fibonacci(2)

fibonacci Method

public static int fibonacci(int n) {
    int fib;
    if (n > 2)
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}

fibonacci(4) fibonacci(3)
fibonacci(2)

Fibonacci Numbers

• The $N$th Fibonacci number is the sum of the previous two Fibonacci numbers
• 0, 1, 1, 2, 3, 5, 8, 13, ...
• Recursive Design:
  – Decomposition & Composition
    • fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  – Base case:
    • fibonacci(1) = 0
    • fibonacci(2) = 1

fibonacci(4) fibonacci(3)
fibonacci(2)
Execution Trace (composition)

\[\text{fibonacci}(4) \rightarrow 2\]

\[\text{fibonacci}(3) \rightarrow 1\]

\[\text{fibonacci}(2) \rightarrow 1\]

\[\text{fibonacci}(1) \rightarrow 0\]

Remember:
Key to Successful Recursion

• if-else statement (or some other branching statement)
• Some branches: recursive call
  – "smaller" arguments or solve "smaller" versions of the same task (decomposition)
  – Combine the results (composition) [if necessary]
• Other branches: no recursive calls
  – stopping cases or base cases

Template ...

\[
\begin{align*}
\text{... method}(\ldots) & \\
\quad \{ & \\
\quad \quad \text{if ( \_ \_)} // \text{base case} & \\
\quad \quad \{ & \\
\quad \quad \} & \\
\quad \quad \text{else} // \text{decomposition} \quad \& \text{composition} & \\
\quad \quad \{ & \\
\quad \quad \} & \\
\quad \text{return \_ \_} ; // \text{if not void method} & \\
\quad \}
\end{align*}
\]
What Happens Here?

```java
public static int factorial(int n)
{
    int fact=1;
    if (n > 1)
        fact = factorial(n) * n;
    return fact;
}
```

Warning: Infinite Recursion May Cause a Stack Overflow Error

- Infinite Recursion
  - Problem not getting smaller (no/bad decomposition)
  - Base case exists, but not reachable (bad base case and/or decomposition)
  - No base case
- Stack: keeps track of recursive calls by JVM (OS)
  - Method begins: add data onto the stack
  - Method ends: remove data from the stack
- Recursion never stops; stack eventually runs out of space
  - Stack overflow error

Mistakes in recursion

- No composition -> ?
- Bad composition -> ?

Number of Zeros in a Number

- Example: 2030 has 2 zeros
- If \( n \) has two or more digits
  - the number of zeros is the number of zeros in \( n \) with the last digit removed
  - plus an additional 1 if the last digit is zero
- Examples:
  - number of zeros in 20030 is number of zeros in 2003 plus 1
  - number of zeros in 20031 is number of zeros in 2003 plus 0

numberOfZeros Recursive Design

- numberOfZeros in the number \( N \)
- \( K \) = number of digits in \( N \)
- Decomposition:
  - numberOfZeros in the first \( K - 1 \) digits
  - Last digit
- Composition:
  - Add:
    - numberOfZeros in the first \( K - 1 \) digits
    - 1 if the last digit is zero
- Base case:
  - \( N \) has one digit (\( K = 1 \))
public static int numberOfZeros(int n) 
{
    int zeroCount;
    if (n==0) 
        zeroCount = 1;
    else if (n < 10)  // and not 0
        zeroCount = 0;  // 0 for no zeros
    else if (n%10 == 0)
        zeroCount = numberOfZeros(n/10) + 1;
    else  // n%10 != 0
        zeroCount = numberOfZeros(n/10);
    return zeroCount;
}

Which is (are) the base case(s)? Why?
Decomposition, Why?
Composition, Why?

public static int numberOfZeros(int n) 
{
    int zeroCount;
    if (n==0) 
        zeroCount = 1;
    else if (n < 10)  // and not 0
        zeroCount = 0;  // 0 for no zeros
    else if (n%10 == 0)
        zeroCount = numberOfZeros(n/10) + 1;
    else  // n%10 != 0
        zeroCount = numberOfZeros(n/10);
    return zeroCount;
}

Execution Trace (decomposition)

Each method invocation will execute one of the if-else cases shown at right.

class Execution Trace (composition)

Recursive calls return

class Number in English Words

• Process an integer and print out its digits in words
  – Input: 123
  – Output: "one two three"
• RecursionDemo class

public static void inWords(int numeral)
{
    if (numeral < 10)
        System.out.print(digitWord(numeral) + " ");
    else //numeral has two or more digits
    {
        inWords(numeral/10);
        System.out.print(digitWord(numeral%10) + " ");
    }
}

inWords method

inWords Recursive Design

• inWords prints a number N in English words
• K = number of digits in N
• Decomposition:
  – inWords for the first K – 1 digits
  – Print the last digit
• Composition:
  – Execution order of composed steps [more later]
• Base case:
  – N has one digit (K = 1)
What Happens with a Recursive Call

• \texttt{inWords} (slightly simplified) with argument 987

```
inWords(987)  
if (987 < 10)  
  // print digit here  
else  //two or more digits left  
  {  
inWords(987/10);  
  // print digit here  
}
```

1. The if condition is false
2. recursive call to \texttt{inWords}, with 987/10 or 98 as the argument
3. Computation waits here until recursive call returns
4. The argument is getting shorter and will eventually get to the base case.
Composition Matters

public static void inWords(int numeral) {
    if (numeral < 10)
        System.out.print(digitWord(numeral) + " ");
    else if (numeral has two or more digits) {
        System.out.print(digitWord(numeral%10) + " ");
        inWords(numeral/10);
    }
}

Recursive Design:
1. Print the last digit
2. inWords for the first K – 1 digits

Execution Trace (decomposition)
inwords(987)
  print “seven”
inwords(98)
  print “eight”
inwords(9)
  print “nine”
Output: seven eight nine
Execution Trace (composition)

inwords(987)

- print “seven” inwords(98)
  - print “eight” inwords(9)
    - print “nine”

No additional output

"Name in the Phone Book" Revisited

Search:
  - middle page = (first page + last page)/2
  - Go to middle page;
  - If (name is on middle page)
    - done //this is the base case
  - else if (name is alphabetically before middle page)
    - last page = middle page // redefine to front half
    - Search / recursive call
  - else // name must be after middle page
    - first page = middle page // redefine to back half
    - Search / recursive call

No additional output

Binary Search Algorithm

- Searching a list for a particular value
  - sequential and binary are two common algorithms
- Sequential search (aka linear search):
  - Not very efficient
  - Easy to understand and program
- Binary search:
  - more efficient than sequential
  - but the list must be sorted first!

Why Is It Called "Binary" Search?

Compare sequential and binary search algorithms:
  - How many elements are eliminated from the list each time a value is read from the list and it is not the "target" value?

Sequential search: only one item
Binary search: half the list

That is why it is called binary - each unsuccessful test for the target value reduces the remaining search list by 1/2.

Binary Search

- public find(target) calls
down private search(target, first, last)
- returns the index of the entry if the target value is found or -1 if it is not found
- Compare it to the pseudocode for the "name in the phone book" problem

private int search(int target, int first, int last)
{
    int location = -1; // not found
    if (first <= last) // range is not empty
    {
        int mid = (first + last)/2;
        if (target == a[mid])
            location = mid;
        else if (target < a[mid]) // first half
            location = search(target, first, mid - 1);
        else
            location = search(target, mid + 1, last);
    }
    return location;
}

Where is the composition?

- If no items
  - not found (-1)
- Else if target is in the middle
  - middle location
- Else
  - location found by search(first half) or search(second half)
Binary Search Example

**target is 33**

The array looks like this:

<table>
<thead>
<tr>
<th>Indexes</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>5 7 9 13 32 33 42 54 56 88</td>
</tr>
</tbody>
</table>

mid = (5 + 9) / 2 (which is 4)

33 > a[mid] (that is, 33 > a[4])

So, if 33 is in the array, then 33 is one of:

5 6 7 8 9

Eliminate half of the remaining elements from consideration because array elements are sorted.

Tips

• Don’t throw away answers (return values) -- need to compose the answers
  – Common programming mistake: not capturing and composing answers (return values)
• Only one return statement at the end
  – Easier to keep track of and debug return values
  – “One entry, one exit”
  – www.cs.fit.edu/~pkc/classes/cse1001/BinarySearch/BinarySearch.java

Worst-case Analysis

• Item not in the array (size N)
• T(N) = number of comparisons with array elements
  – T(1) = 1
  – T(N) = 1 + T(N/2)

= 1 + [1 + T(N/4)]

= 1 + [1 + [1 + T(N/8)]]
Worst-case Analysis

- Item not in the array (size $N$)
- $T(N)$ = number of comparisons with array elements
- $T(1) = 1$
- $T(N) = 1 + T(N/2)$
  $= 1 + [1 + T(N/4)]$
  $= 2 + T(N/4)$
  $= 2 + [1 + T(N/8)]$
  $= 3 + T(N/8)$
  $= \ldots$

Main steps for analysis

- Set up the recurrence equations for the recursive algorithm
- Expand the equations a few times
- Look for a pattern
- Introduce a variable to describe the pattern
- Find the value for the variable via the base case
- Get rid of the variable via substitution

Binary vs. Sequential Search

- Binary Search
  - $\log_2 N + 1$ comparisons (worst case)
- Sequential/Linear Search
  - $N$ comparisons (worst case)
- Binary Search is faster but
  - array is assumed to be sorted beforehand
  - Faster searching algorithms for "non-sorted arrays"
  - More sophisticated data structures than arrays
  - Later courses

Recursive Versus Iterative Methods

- All recursive algorithms/methods can be rewritten without recursion.
- Iterative methods use loops instead of recursion
- Iterative methods generally run faster and use less memory--less overhead in keeping track of method calls
So When Should You Use Recursion?

- Solutions/algorithms for some problems are inherently recursive
  - iterative implementation could be more complicated
- When efficiency is less important
  - it might make the code easier to understand
- **Bottom line is about:**
  - Algorithm design
  - Tradeoff between readability and efficiency

### Merge Sort—
A Recursive Sorting Algorithm

- Example of *divide and conquer* algorithm
- Recursive design:
  - Divides array in half and *merge sorts* the halves (*decomposition*)
  - Combines two sorted halves (*composition*)
  - Array has only one element (*base case*)
- Harder to implement iteratively

### Execution Trace (decomposition)

```
3 6 8 2 5 4 7 1
```

### Execution Trace (composition)

```
1 2 3 4 5 6 7 8
```

### Merging Two Sorted Arrays

```
2 3 6 8
3 6 2 8
3 6 8 2
```

**Pages 807** *NOT a good tip

*Programming Tip: Ask Until the User Gets It Right*  

- Recursion continues until user enters valid input.

```
public void getCount()
{
    System.out.println("Enter a positive number.");
    count = SavitchIn.readLineInt();
    if (count <= 0)
    {
        System.out.println("Input must be positive.");
        System.out.println("Try again.");
        getCount(); //start over
    }
}
```

- No notion of a smaller problem for recursive design
- Easily implemented using iteration without loss of readability
Merging Two Sorted Arrays

1. If array a has more than one element:
   a. Copy the first half of the elements in a to array front
   b. Copy the rest of the elements in a to array tail
   c. Merge Sort front
   d. Merge Sort tail
   e. Merge the elements in front and tail into a
   Otherwise, do nothing

Recursive calls

make "smaller" problems by dividing array

Combine the two sorted arrays

Merging Two Sorted Arrays

1. If array a has more than one element:
   a. Copy the first half of the elements in a to array front
   b. Copy the rest of the elements in a to array tail
   c. Merge Sort front
   d. Merge Sort tail
   e. Merge the elements in front and tail into a
   Otherwise, do nothing

Recursive calls

merge(a, front, tail)

sort(front);
sort(tail);

base case: a.length == 1 so a is sorted and no recursive call is necessary.

Worst-case Theoretical Analysis

- Comparisons of array elements
- None during decomposition
- Only during merging two sorted arrays (composition)
  - To get an array of size N from two sorted arrays of size N/2
  - N - 1 comparisons (worst case: the largest two elements are in different halves)
Analysis: Array of size \( N \)

- Let \( T(N) \) be the number of comparisons
- \( T(1) = 0 \)
- \( T(N) = 2 \ T(N/2) + (N - 1) \)

\[
T(1) = 0 \\
T(N) = 2 \ T(N/2) + (N - 1)
\]

\[
= 2 \cdot 2 \ T(N/4) + (N/2 - 1) + (N - 1) \\
= 4 \ T(N/4) + (N - 2) + (N - 1) \\
= 4 \cdot 2 \ T(N/8) + (N/4 - 1) + (N - 2) + (N - 1) \\
= 8 \ T(N/8) + (N - 4) + (N - 2) + (N - 1) \\
= 8 \ T(N/8) + 3N - (1 + 2 + 4)
\]

\[
= 2^k \ T(N/2^k) + kN - (1 + 2 + \ldots + 2^{k-1})
\]
Analysis Continued

- $T(N) = 2^k T(N/2^k) + kN - (1 + 2 + \ldots + 2^{k-1})$ \[1\]
  \[= 2^k T(N/2^k) + kN - (2^k - 1) \[2\]
- $T(N/2^k)$ gets smaller until the base case $T(1)$:
  \[- 2^k = N\]
  \[- k = \log_2 N\]
- Replace terms with $k$ in \[2\]:
  \[T(N) = N T(N/N) + \log_2 N^N - (N - 1)\]
  \[= N T(1) + \log_2 N - (N - 1)\]
  \[= \log_2 N - N + 1\]
  \["MogN" algorithm\]

Geometric Series and Sum

- $1 + 2 + 4 + \ldots + 2^k$
  \[- 1 + 2 = 3 \quad (4 - 1)\]
  \[- 1 + 2 + 4 = 7 \quad (8 - 1)\]
  \[- 1 + 2 + 4 + 8 = 15 \quad (16 - 1)\]

Geometric Series and Sum

- $1 + 2 + 4 + \ldots + 2^k$
  \[- 1 + 2 = 3 \quad (4 - 1)\]
  \[- 1 + 2 + 4 = 7 \quad (8 - 1)\]
  \[- 1 + 2 + 4 + 8 = 15 \quad (16 - 1)\]

- $1 + r + r^2 + \ldots + r^k$
  \[= (r^{k+1} - 1) / (r - 1) \quad [\text{for } r > 1]\]

Merge Sort Vs. Selection/Insertion/Bubble Sort

- Merge Sort
  - "MogN" algorithm (in comparisons)
- Selection/Insertion/Bubble Sort
  - "N" algorithm (in comparisons)
- "MogN" is "optimal" for sorting
  - Proven that the sorting problem cannot be solved with fewer comparisons
  - Other MogN algorithms exist, many are recursive

Real Data Set: Web Server Log

- 4.6 MB (44057 entries)
- Example entry in log:
- Extracted features
  - remote-host names (strings)
  - file-size (integers)
- List size - 100 to 44000 entries
Google’s PageRank (1998)

- PageRank(x) depends on:
  1. How many pages (y’s) linking to x
      • how many incoming links (citations) from y’s to x
  2. How important those pages (y’s) are:
      • PageRank(y)’s

- How to determine PageRank(y)’s?
- What is the base case?

Summary

- Binary Search
  - Given an ordered list
  - “logN” algorithm (in comparisons)
  - “Optimal”
- Merge Sort
  - Recursive sorting algorithm
  - “NlogN” algorithm (in comparisons)
  - “Optimal”