RECURSION

1

The Recursion Pattern

- Recursion: is a technique by which a method makes one or more calls to itself during execution
- Classic example – the factorial function:

  n! = 1· 2· 3· ··· · (n-1)· n

- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

2

Recursion Factorial - Java

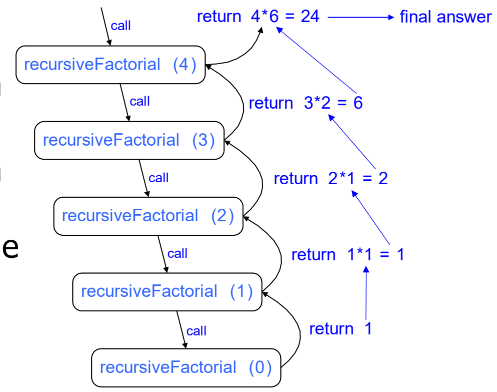
- As a Java method:

```
public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0) // argument must be nonnegative.
        throw new IllegalArgumentException( );

    if (n == 0)
        return 1; // base case

    return n * factorial(n-1); // recursive case
}
```

3

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



4

Binary Search

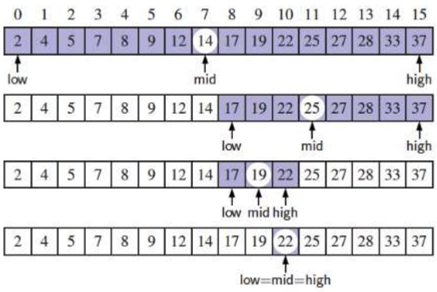
- Binary search is an efficient algorithm for finding an item from a **sorted list of items.** It works by repeatedly dividing in half the portion of the list that could contain the item, until the list is narrowed down the possible locations to just one.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

5

Visualizing Binary Search

- Middle is identified as mid = ⌊(low+high)/2⌋
- There are three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



6

Binary Search

Search for an integer in an ordered list

```
/**
 * Returns true if the target value is found in the indicated portion of the
data array.
 * This search only considers the array portion from data[low] to data[high]
inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
    if (low > high)
        return false; // interval empty; no match
    else {
        int mid = (low + high) / 2;
        if (target == data[mid])
            return true; // found a match
        else if (target < data[mid])
             // recur left of the middle
            return binarySearch(data, target, low, mid - 1);
        else
            // recur right of the middle
            return binarySearch(data, target, mid + 1, high);
    }
}
```

7

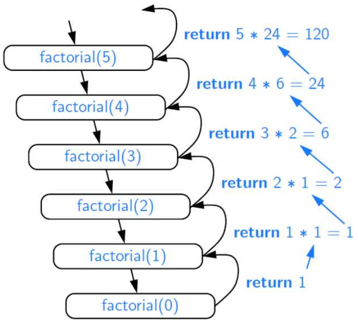
Analyzing Recursive Algorithms

- Mathematical techniques for analyzing the efficiency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm.

8

Analysis of Computing Factorials

- A sample recursion trace for our factorial method was given on the right.
- To compute factorial(n), we see that there are a total of n+1 activations, as the parameter decreases from n in the first call, to n−1 in the second call, and so on, until reaching the base case with parameter 0.
- Each individual activation of factorial executes a constant number of operations.
- Therefore, we conclude that the overall number of operations for computing factorial(n) is O(n), as there are n+1 activations, each of which accounts for O(1) operations.



9

Analyzing Binary Search

- The remaining portion of the list is of size high – low + 1
- After one comparison, this becomes one of the following:

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\leq\frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\leq\frac{\text{high}-\text{low}+1}{2}$$

- Thus, each recursive call divides the search region in half; hence, there can be at most log n levels so runs in O(log n) time

10

Further Examples of Recursion

- If a recursive call starts at most one other, we call this a ***linear recursion.***
- If a recursive call may start two others, we call this a ***binary recursion.***
- If a recursive call may start three or more others, this is ***multiple recursion.***

11

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform **a single recursive call**
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

12

Example of Linear Recursion

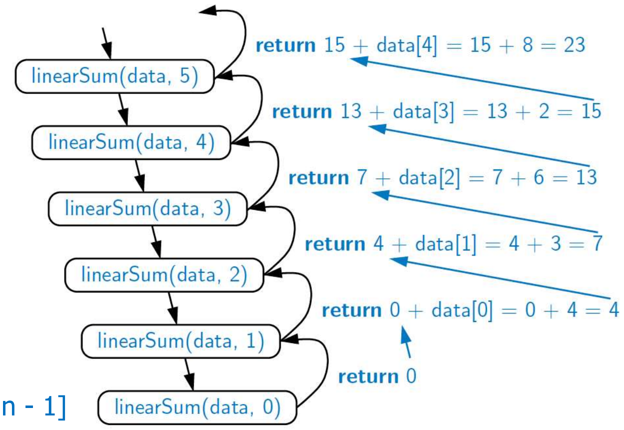
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

Computing the sum of a sequence recursively, by adding the last number to the sum of the first n−1.

Algorithm linearSum(A, n):
Input:
  Array, A, of integers
  Integer n such that
    0 ≤ n ≤ |A|
Output:
    Sum of the first n integers in A

if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]

Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]



13

Reversing an Array

Algorithm reverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at

if i < j then
    Swap A[i] and A[j]
    reverseArray(A, i + 1, j – 1)
return

0	1	2	3	4	5	6	7
4	3	6	2	7	8	9	5
5	3	6	2	7	8	9	4
5	9	6	2	7	8	3	4
5	9	8	2	7	6	3	4
5	9	8	7	2	6	3	4

A trace of the recursion for reversing a sequence. The highlighted portion has yet to be reversed.

14

Designing Recursive Algorithm

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

15

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
/** Reverses the contents of subarray data[low] through data[high]
inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
    if (low < high) { // if at least two elements in subarray
        int temp = data[low]; // swap data[low] and data[high]
        data[low] = data[high];
        data[high] = temp;
        reverseArray(data, low + 1, high - 1); // recur on the rest
    }
}
```

16

Recursive Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in **O(n)** time (for we make n recursive calls)
- We can do better than this, however

```
/** Computes the value of x raised to the nth power, for
nonnegative integer n. */
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
}
```

17

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

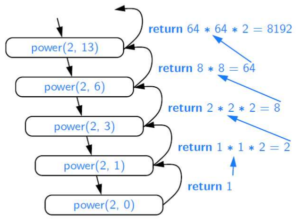
18

Recursive Squaring Method

Algorithm Power(x, n):
  Input: A number x and integer n = 0
  Output: The value x^n
 if n = 0 **then**
   return 1
 if n is odd **then**
   y = Power(x, (n - 1)/ 2)
   return x · y ·y
 else
   y = Power(x, n/ 2)
   return y · y

19

Analysis



Algorithm Power(x, n):
  Input: A number x and integer n = 0
  Output: The value x^n
 if n = 0 **then**
   return 1
 if n is odd **then**
   y = Power(x, (n - 1)/ 2)
   return x · y · y
 else
   y = Power(x, n/ 2)
   return y · y

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

20

Eliminating Recursion

- The main benefit of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems.
- By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.
- In general, we can use the stack data structure to convert a recursive algorithm into a non-recursive algorithm.

21

Eliminating Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):
  Input: An array A and nonnegative integer indices i and j
  Output: The reversal of the elements in A starting at index i and ending at j
  while i < j **do**
    Swap A[i] and A[j]
    i = i + 1
    j = j - 1
  return

22

Ex: A Nonrecursive Implementation of Binary Search

```java
/** Returns true if the target value is found in the data array. */
public static boolean binarySearchIterative(int[ ] data, int target) {
    int low = 0;
    int high = data.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (target == data[mid]) // found a match
            return true;
        else if (target < data[mid])
            high = mid - 1; // only consider values left of mid
        else
            low = mid + 1; // only consider values right of mid
    }
    return false; // loop ended without success
}
```

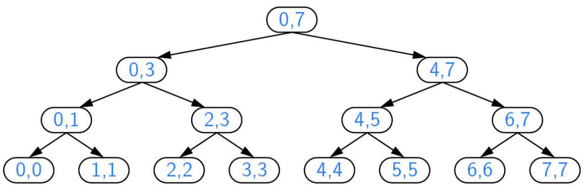
23

Binary Recursive Method

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case
- Problem: add all the numbers in an integer array A:

```java
public static int binarySum(int[ ] data, int low, int high) {
if (low > high) // zero elements in subarray
return 0;
else if (low == high) // one element in subarray
return data[low];
else {
int mid = (low + high) / 2;
return binarySum(data, low, mid) + binarySum(data, mid+1, high);
}
}
```

Example trace:



24

Multiple Recursion

- Motivating example:
  - summation puzzles
    - *pot + pan = bib*
- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

pop
pan
bib

p = ?
o = ?
a = ?
n = ?
b = ?
i = ?

25

Algorithm for Multiple Recursion

pot
pan
bib

```
Algorithm PuzzleSolve(k,S,U):
 Input: Integer k for the length of sequence, sequence S, and set U (universe of
    elements to test)
 Output: Enumeration of all k-length extensions to S using elements in U without
    repetitions
 for all e in U do
    Remove e from U          {e is now being used}
    Add e to the end of S
    if k = 1 then
         Test whether S is a configuration that solves the puzzle
         if S solves the puzzle then
                return "Solution found: " S
    else
         PuzzleSolve(k - 1, S,U)
    Add e back to U          {e is now unused}
    Remove e from the end of S
```

26

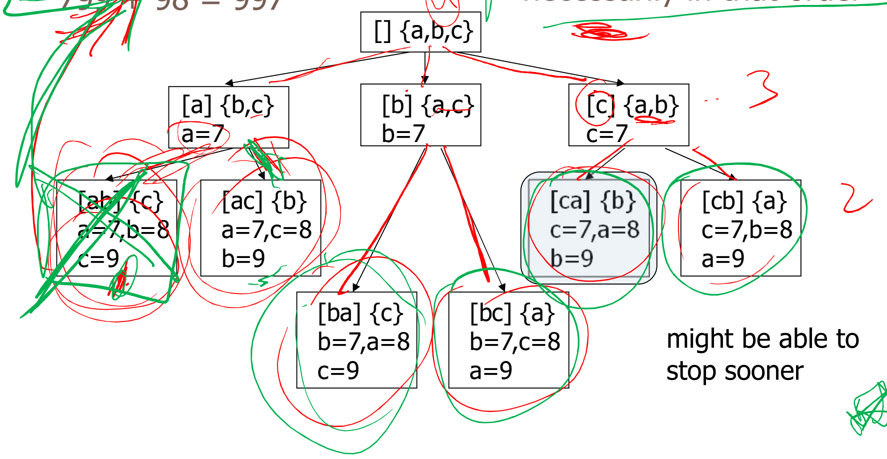
Example

[a b c]

cbb + ba = abc

799 + 98 = 997

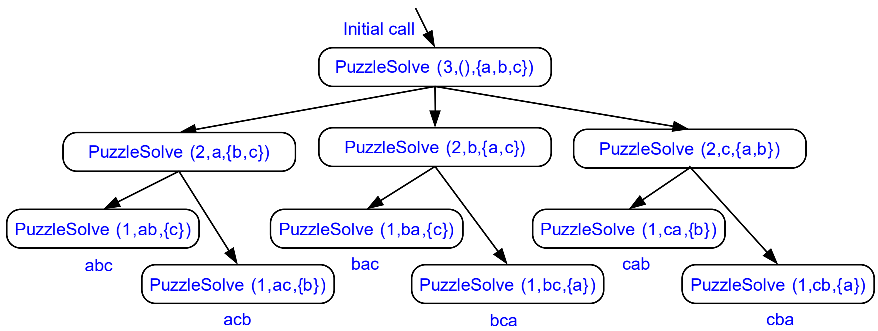
a,b,c stand for 7,8,9; not necessarily in that order



might be able to stop sooner

27

Visualizing PuzzleSolve



28

Exercise

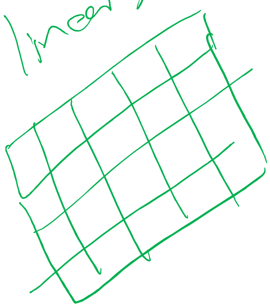
Exam

- Describe (Pseudocode) a way to use recursion to compute the sum of all the elements in an n×n (two-dimensional) array of integers. What is your running time and space usage?

0

linear sum

recursive sum (A, . . .)



n×n

29

Additional Example

30

An Inefficient Recursion for Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  $F_0 = 0$
  $F_1 = 1$
  $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

  Algorithm BinaryFib(k):
  Input: Nonnegative integer k
  Output: The kth Fibonacci number F_k
  if $k = 1$ **then**
  return k
  else
  return BinaryFib($k - 1$) + BinaryFib($k - 2$)

31

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

32

A Better Fibonacci Algorithm

- Use linear recursion instead

Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k , F_{k-1})
 if k = 1 **then**
    return (k, 0)
 else
    (i, j) = LinearFibonacci(k – 1)
    return (i +j, i)

- LinearFibonacci makes k–1 recursive calls

33

Exercise

- Write a program for solving summation puzzles by enumerating and testing all possible configurations. Using your program, solve the **three** different puzzles given
  - *pot* + *pan* = *bib*
  - *dog* + *cat* = *pig*
  - *boy* + *girl* = *baby*

 where each char is a digit.

34