Recursion

Dr. Nguyễn Thanh Hung

1

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

2

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

3

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
```

4

Overview

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

5

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

6

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

7

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

8

Factorial (*N*!)

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

9

`factorial` Method

```
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;
}
```

10

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

11

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

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

12

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

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

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

13

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

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

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

14

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

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

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

15

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

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

16

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

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

17

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

18

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)

factorial(3)   4

19

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)

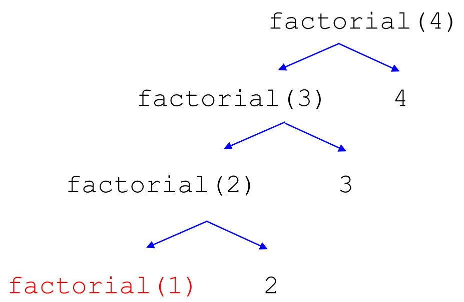
factorial(3)   4

factorial(2)   3

20

Execution Trace (decomposition)

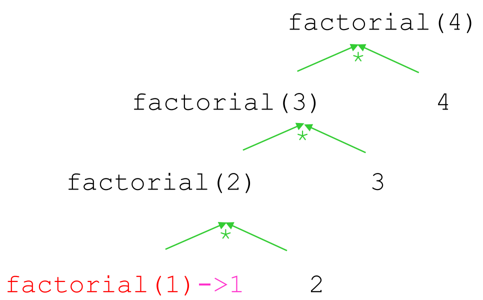
```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



21

Execution Trace (composition)

```
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;
}
```



22

Execution Trace (composition)

```
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)
                 *
     factorial(3)        4
            *
factorial(2)->2      3
```

23

Execution Trace (composition)

```
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)
                 *
     factorial(3)->6     4
```

24

Execution Trace (composition)

```
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`

25

Improved `factorial` Method

```
public static int factorial(int n)
{
  int fact=1;  // base case value

  if (n > 1)   // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  // else do nothing; base case

  return fact;
}
```

26

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

27

`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;
}
```

28

Execution Trace (decomposition)

```
            fibonacci(4)

fibonacci(3)            fibonacci(2)
```

29

Execution Trace (decomposition)

```
                        fibonacci(4)

            fibonacci(3)            fibonacci(2)

fibonacci(2)        fibonacci(1)
```

30

Execution Trace (composition)

```
              fibonacci(4)
                   +
      fibonacci(3)         fibonacci(2)
            +
fibonacci(2)->1   fibonacci(1)->0
```

31

Execution Trace (composition)

```
              fibonacci(4)
                   +
      fibonacci(3)->1      fibonacci(2)->1
```

32

Execution Trace (composition)

```
fibonacci(4)->2
```

33

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

34

Template

```
… method(…)
{
   if ( … )// base case
   {
   }
   else // decomposition & composition
   {
   }
   return … ; // if not void method
}
```

35

Template (only one base case)

```
… method(…)
{
   … result = …   ;//base case

   if ( … ) // not base case
   { //decomposition & composition
     result = …
   }

   return result;
}
```

36

What Happens Here?

```
public static int factorial(int n)
{
  int fact=1;

  if (n > 1)
    fact = factorial(n) * n;

  return fact;
}
```

37

What Happens Here?

```
public static int factorial(int n)
{
  return factorial(n - 1) * n;
}
```

38

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**

39

Mistakes in recursion

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

40

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

recursive

41

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* - 1digits
    - 1 if the last digit is zero
- Base case:
  - *N* has one digit (*K* = 1)

42

`numberOfZeros` method

```
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?

Decompostion, Why?

Composition, why?

43

Execution Trace (decomposition)

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

```
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;
}
```

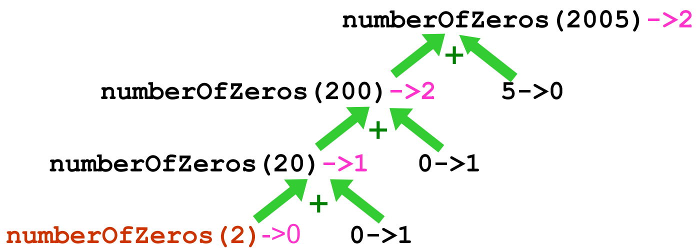
```
                    numberOfZeros(2005)
           numberOfZeros(200)          5
      numberOfZeros(20)            0
numberOfZeros(2)            0
```

44

Execution Trace (composition)

Recursive calls return

```
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;
}
```



45

Number in English Words

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

46

inWords Resursive 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)

47

`inWords` method

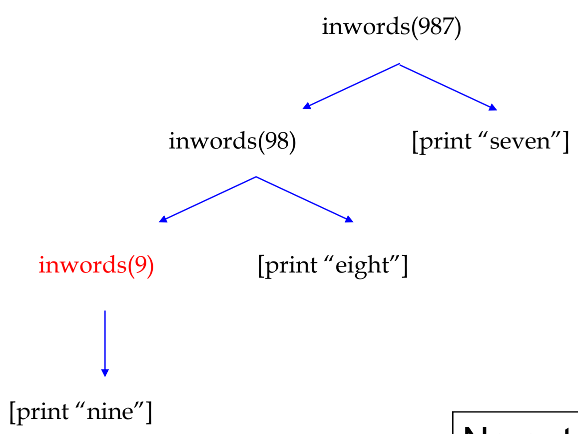
Base case executes when only 1 digit is left

Size of problem is reduced for each recursive call

```
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) + " ");
    }
}
```

48

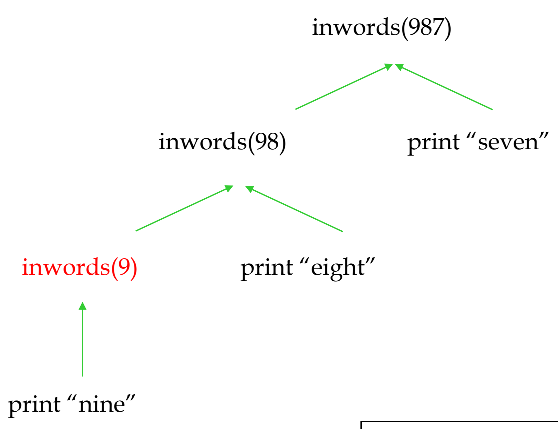
Execution Trace (decomposition)



No output yet, why?

49

Execution Trace (composition)



Output: nine eight seven

50

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

1

What Happens with a Recursive Call

- `inWords` (slightly simplified) with argument `987`

51

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

2

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print digit here
}
```

Execution Trace

The argument is getting shorter and will eventually get to the base case.

Computation waits here until recursive call returns

- The `if` condition is false
- recursive call to `inWords`, with `987/10` or `98` as the argument

52

Execution Trace

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

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print digit here
}
```

3

```
inWords(9)
if (9 < 10)
    // print digit here
else //two or more digits left
{
   inWords(numeral/10);
   // print digit here
}
```

- the if condition is false
- another recursive call is made.

53

Execution Trace

Output: nine

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

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print 98 % 10
}
```

4

```
if (9 < 10)
    // print nine
else //two or more digits left
{
   inWords(numeral/10);
   // print digit here
}
```

- if condition is true (base case)
- prints nine and returns
- no recursive call

54

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

Execution Trace

5

```
if (98 < 10)
    // print out digit here
else //two or more digits left
{
   inWords(98/10);
   // print out 98 % 10 here
}
```

Output: `nine eight`

- executes the next statement after the recursive call
- prints eight and then returns

55

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

6

Execution Trace

Output: `nine eight seven`

- executes the next statement after the recursive method call.
- prints seven and returns

56

Composition Matters

Print before making the recursive call

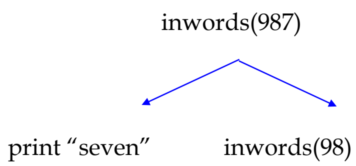
```
public static void inWords(int numeral)
{
   if (numeral < 10)
      System.out.print(digitWord(numeral) + " ");
   else //numeral has two or more digits
   {
      System.out.print(digitWord(numeral%10) + " ");
      inWords(numeral/10);
   }
}
```

Recursive Design:

1. Print the last digit
2. inWords for the first *K* – 1 digits

57

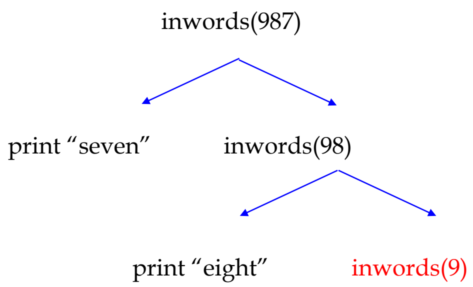
Execution Trace (decomposition)



Output: seven

58

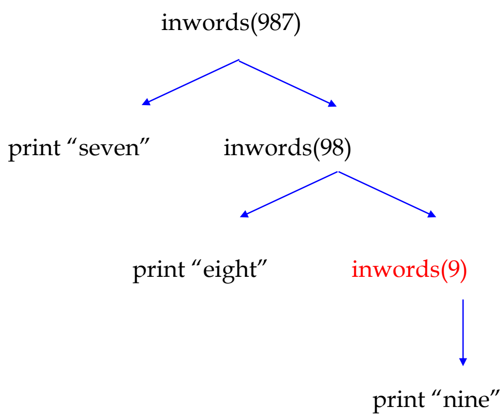
Execution Trace (decomposition)



Output: seven eight

59

Execution Trace (decomposition)



Output: seven eight nine

60

Execution Trace (composition)

inwords(987)

print “seven” inwords(98)

print “eight” inwords(9)

print “nine”

No additional output

61

"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
```

62

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!*

63

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.

64

Binary Search Method

- `public find(target)` calls `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 //(target > a[mid]) second half
          location = search(target, mid + 1, last);
    }

    return location;
 }
```

65

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)

66

Binary Search Example

`target` is `33`

The array **a** looks like this:

Indices	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

`mid = (0 + 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
					33	42	54	56	88

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

67

Binary Search Example

`target` is `33`

The array **a** looks like this:

Indexes	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

`mid = (5 + 9) / 2` (which is `7`)

`33 < a[mid]` (that is, `33 < a[7]`)

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

					5	6			
					33	42			

Eliminate half of the remaining elements

`mid = (5 + 6) / 2` (which is `5`)

`33 == a[mid]`

So we found `33` at index `5`:

					5				
					33				

68

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

69

Worst-case Analysis

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

70

Worst-case Analysis

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

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

71

Worst-case Analysis

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

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

  $= 2 + T(N / 4)$

  $= 2 + [1 + T(N / 8)]$

72

Worst-case Analysis

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

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

  $= 2 + T(N / 4)$ ←

  $= 2 + [1 + T(N / 8)]$

  $= 3 + T(N / 8)$ ←

  $= \ldots$

73

Worst-case Analysis

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

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

  $= 2 + T(N / 4)$ ←

  $= 2 + [1 + T(N / 8)]$

  $= 3 + T(N / 8)$ ←

  $= \ldots$

  $= k + T(N / 2^k)$ [1]

74

Worst-case Analysis

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

$$T(N) = \log_2 N + T(N / N)$$
$$= \log_2 N + T(1)$$
$$= \log_2 N + 1$$

- "$\log_2 N$" algorithm
- We used *recurrence equations*

75

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

76

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

77

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

78

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*

79

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
   }
}
```

read a number

Use a recursive call to get another number.

- *No notion of a smaller problem for recursive design*
- *Easily implemented using iteration without loss of readability*

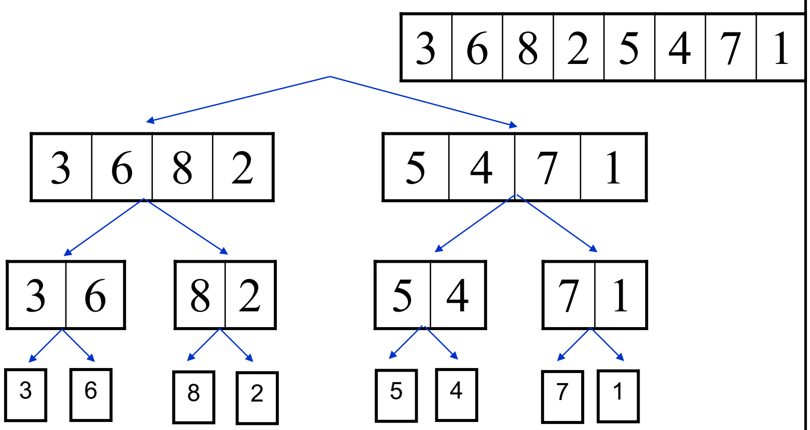
80

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

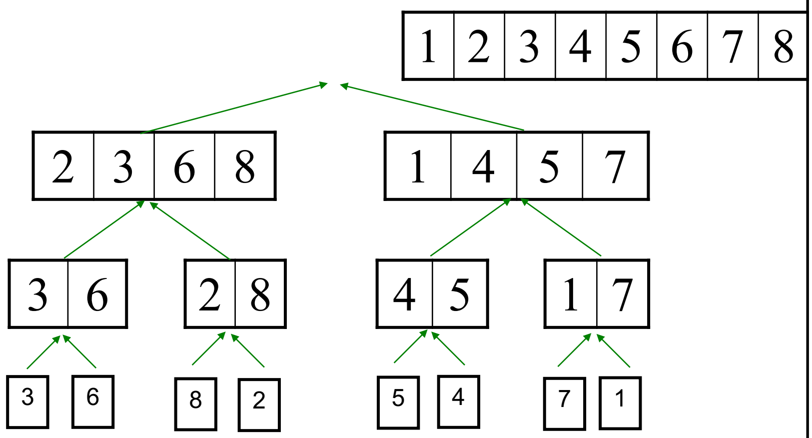
81

Execution Trace (decomposition)



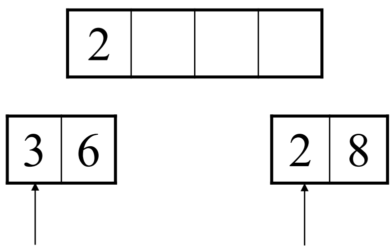
82

Execution Trace (composition)



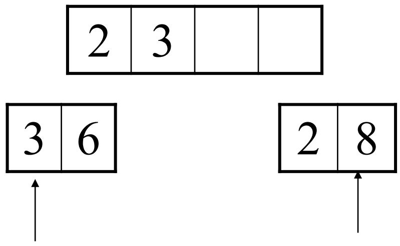
83

Merging Two Sorted Arrays



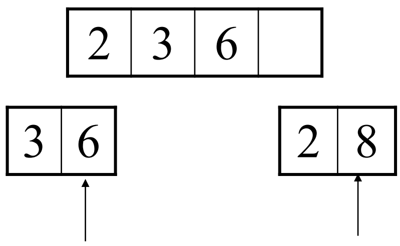
84

Merging Two Sorted Arrays



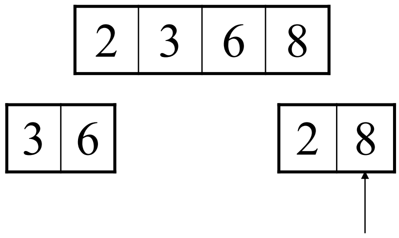
85

Merging Two Sorted Arrays



86

Merging Two Sorted Arrays



87

Merge Sort Algorithm

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*
2. Otherwise, do nothing

88

Merge Sort

```
public static void sort(int[] a)
{
   if (a.length >= 2)
   {
      int halfLength = a.length / 2;
      int[] front = new int[halfLength];
      int[] tail = new int[a.length – halfLength];
      divide(a, front, tail);
      sort(front);
      sort(tail);
      merge(a, front, tail);
   }
   // else do nothing.
}
```

do recursive case if true, base case if false

recursive calls

make "smaller" problems by dividing array

Combine the two sorted arrays

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

89

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)

90

Analysis: Array of size N

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

91

Analysis: Array of size N

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

  $= 2\ [2\ T(N / 4) + (N / 2 - 1)] + (N - 1)$

92

Analysis: Array of size N

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

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

93

Analysis: Array of size N

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

$$= 2\ [2\ T(N / 4) + (N / 2 - 1)] + (N - 1)$$
$$= 4\ T(N / 4) + (N - 2) + (N - 1) \quad \leftarrow$$
$$= 4\ [\ 2\ T(N / 8) + (N / 4 - 1)\ ] + (N - 2) + (N - 1)$$
$$= 8\ T(N / 8) + (N - 4) + (N - 2) + (N - 1) \quad \leftarrow$$

94

Analysis: Array of size N

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

$= 2\ [2\ T(N / 4) + (N / 2 - 1)] + (N - 1)$

$= 4\ T(N / 4) + (N - 2) + (N - 1)$ ←

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

$= 8\ T(N / 8) + (N - 4) + (N - 2) + (N - 1)$ ←

$= 8\ T(N / 8) + 3N - (1 + 2 + 4)$

95

Analysis: Array of size N

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

$= 2\ [2\ T(N / 4) + (N / 2 - 1)] + (N - 1)$

$= 4\ T(N / 4) + (N - 2) + (N - 1)$ ←

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

$= 8\ T(N / 8) + (N - 4) + (N - 2) + (N - 1)$ ←

$= 8\ T(N / 8) + 3N - (1 + 2 + 4)$

$= \ldots$

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

96

Analysis Continued

- $T(N) = 2^k T(N / 2^k) + kN – (1 + 2 + \dots 2^{k-1})$ [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]:
  $T(N) = N\,T(N / N) + \log_2 N * N – (N – 1)$
  $= N\,T(1) + N\log_2 N – (N – 1)$
  $= N\log_2 N – N + 1$
- "$N\log_2 N$" algorithm

97

Geometric Series and Sum

- $1 + 2 + 4 + 8 + \dots + 2^k$
  - $1 + 2 = 3$
  - $1 + 2 + 4 = 7$
  - $1 + 2 + 4 + 8 = 15$

98

Geometric Series and Sum

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

99

Geometric Series and Sum

- $1 + 2 + 4 + 8 + \ldots + 2^k$
  - $1 + 2 = 3$ $(4 – 1)$
  - $1 + 2 + 4 = 7$ $(8 – 1)$
  - $1 + 2 + 4 + 8 = 15$ $(16 – 1)$
- $1 + 2 + 4 + 8 + \ldots + 2^k$
  $= 2^{k+1} - 1$
- $1 + r + r^2 + r^3 + \ldots + r^k$
  $= r^0 + r^1 + r^2 + r^3 + \ldots + r^k$
  $= (r^{k+1} – 1) / (r – 1)$ [for $r > 1$]

100

Merge Sort Vs. Selection/Insertion/Bubble Sort

- Merge Sort
  - "*N*log*N*" algorithm (in comparisons)
- Selection/Insertion/Bubble Sort
  - "N^2" algorithm (in comparisons)
- "*N*log*N*" is "optimal" for sorting
  - Proven that the sorting problem cannot be solved with fewer comparisons
  - Other *N*log*N* algorithms exist, many are recursive

101

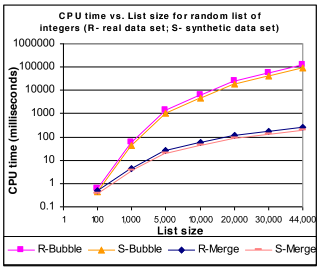
Real Data Set: Web Server Log

- http://www.cs.fit.edu/~pkc/classes/writing/data/jan99.log
- 4.6 MB (44057 entries)
- Example entry in log:

  ip195.dca.primenet.com - - [04/Jan/1999:09:16:51 -0500] "GET / HTTP/1.0" 200 762
- Extracted features
  - remote-host names (strings)
  - file-size (integers)
- List size - 100 to 44000 entries

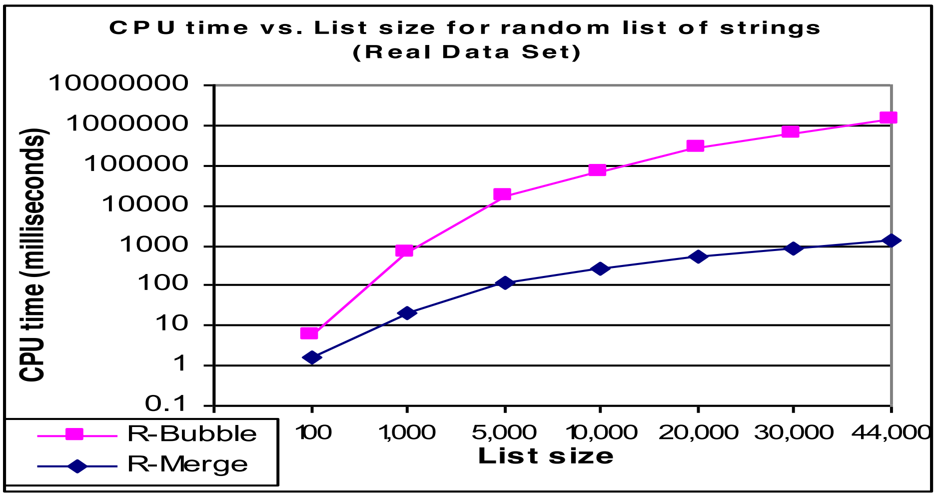
102

CPU Time: Randomly Ordered Integers



103

CPU Time: Randomly Ordered Strings



104

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?

105

Summary

- *Recursive call*: a method that calls itself
- Powerful for algorithm design at times
- Recursive algorithm design:
    - Decomposition (smaller identical problems)
    - Composition (combine results)
    - Base case(s) (smallest problem, no recursive calls)
- Implementation
  - Conditional (e.g. if) statements to separate different cases
  - Avoid infinite recursion
    - Problem is getting smaller (decomposition)
    - Base case exists and reachable
  - Composition could be tricky

106

Summary

- Binary Search
  - Given an ordered list
  - "log*N*" algorithm (in comparisons)
  - "Optimal"
- Merge Sort
  - Recursive sorting algorithm
  - *"N*log*N"* algorithm (in comparisons)
  - "Optimal"

107