NC STATE UNIVERSITY

Recursion

Readings:
Zybooks Chapter 16

Slides are modified from those provided by Marty Stepp
www.buildingjavaprograms.com
Some materials from Joseph Hollingsworth and Murali Sitaraman

NC STATE UNIVERSITY

Recursion

- **Recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.
- **Recursive Programming**: Writing methods that call themselves to solve problems recursively.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems

NC STATE UNIVERSITY

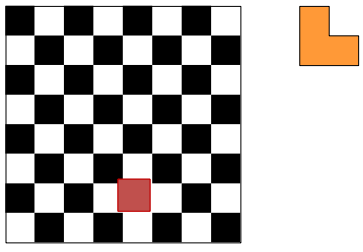
Why learn recursion?

- Some important data structures are inherently recursive (e.g., lists and search trees: coming soon!).
- Some algorithms are inherently recursive (e.g., "divide and conquer" algorithms like mergesort and binary search: also coming soon!).
- Many "functional" programming languages (e.g., Scheme, ML, and Haskell) use recursion exclusively (no loops).
- Leads to elegant, simplistic, short code (when used well).

NC STATE UNIVERSITY

Example: Trominos Puzzle

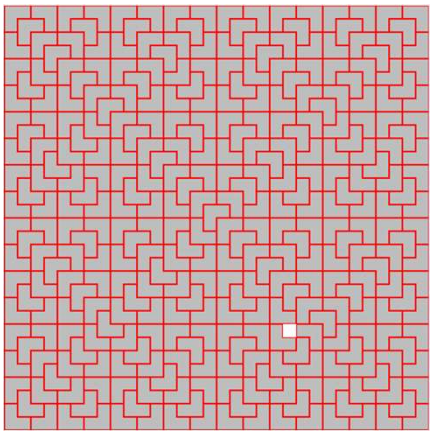
- Suppose you are given a $2^i \times 2^i$ chessboard, for some integer $i \geq 1$, with a single tile already covered (it could be any tile, you don't get to choose).
- Can you cover the remaining tiles exclusively using "trominos" (three tiles in an L shape)?
- The answer is yes, but how?



NC STATE UNIVERSITY

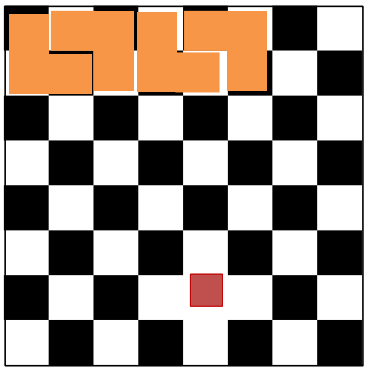
Example: Trominos Puzzle

- For example, here is a solution to a $2^5 \times 2^5$ board.



NC STATE UNIVERSITY

Example: Trominos Puzzle

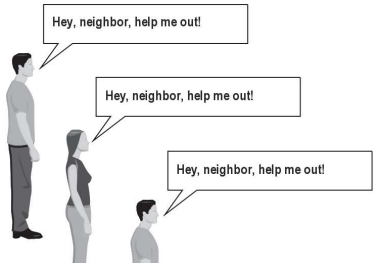


- Looking at a $2^3 \times 2^3$ board, you might be tempted to just iterate across the board.
- But how do you know you won't get stuck later?
- And how would you program all of the possible cases, depending on where the covered tile is?

NC STATE UNIVERSITY

The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?

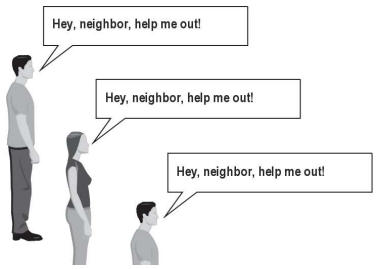


8

NC STATE UNIVERSITY

Recursive algorithm

- Number of people behind me:
  - If there is someone ahead of me,
    ask them how many people are in front of them.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is no one ahead of me, I will answer **0**.



9

NC STATE UNIVERSITY

Recursion and cases

- Every recursive algorithm involves at least 2 cases (these can be implicit):
  - **Base Case**: A simple occurrence that can be answered directly.
  - **Recursive Case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

10

NC STATE UNIVERSITY

Recursion and cases

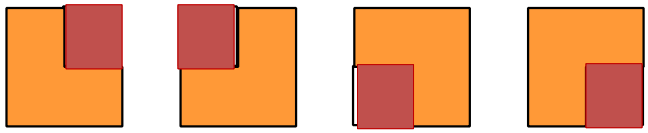
- Every recursive algorithm involves at least 2 cases (these can be implicit):
  - **Base Case**: A simple occurrence that can be answered directly.
  - **Recursive Case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
- Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
- A crucial part of recursive programming is identifying these cases.

11

NC STATE UNIVERSITY

Trominos Puzzle - Recursive Tiling

- Thinking back to our chessboard and trominos, the base case of a $2^1 \times 2^1$ board with a single tile covered is easy: there are exactly four possibilities.

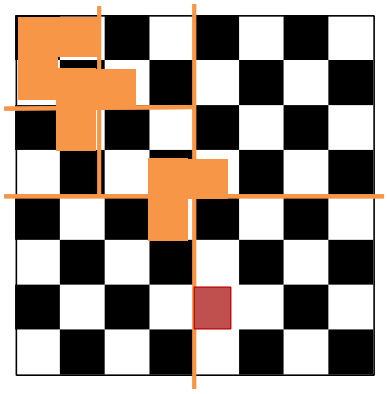


- For the recursive case, we want to tile a $2^i \times 2^i$ board (with a single tile covered) by breaking it up into four different $2^{i-1} \times 2^{i-1}$ boards (each with a single tiled covered).

12

NC STATE UNIVERSITY

Trominos Puzzle - Recursive Tiling



- That just means splitting the chessboard into the quadrants.
- To ensure that every subproblem has a single tile covered, we can place a single tromino in the center.
- That's the entire recursive algorithm! But how do we actually program recursively?

13

NC STATE UNIVERSITY

Recursive Programming – Big Ideas

- A recursive function calls itself in its definition.
  - Suppose we have a function called **`function()`**
- To end in finite time, linear time, we should be parameterize the function by problem size, like **`n`** (e.g., **`function(n)`**):
  - First check for the base case (when **`n`** is small), and return the answer **without** recursive calls.
  - Otherwise, make one or more calls to **`function(m)`** where **`m`** < **`n`**, in order to return the solution to **`function(n)`**.

```
int function(int n) {
   if (…) { return; } //base case
   else { return fun(n-1); } //recursive case
}
```


NC STATE UNIVERSITY

Recursion in Java

- Consider the following method to print a line of * characters:

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();   // end the line of output
}
```

- Write a recursive version of this method (that calls itself).
  - Solve the problem without using any loops.

A base case

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```


Handling more cases

- Handling additional cases, with no loops (in a bad way):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...
}
```

Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1);    // prints "*"
    } else if (n == 3) {
        System.out.print("*");
        printStars(2);    // prints "**"
    } else if (n == 4) {
        System.out.print("*");
        printStars(3);    // prints "***"
    } else ...
}
```


Using recursion properly

- Condensing the recursive cases into a single case:

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

NC STATE UNIVERSITY

"Recursion Zen"

- The real, even simpler, base case is an `n` of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

  - **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

NC STATE UNIVERSITY

Extra recursive trace

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

  - What is the result of the following call?

```
mystery(648)
```

A recursive trace

- Consider the following recursive method:

```
public static int mystery2(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery2(n / 10);
        int b = mystery2(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?

```
mystery2(348)
```


Exercise

- Write a recursive method `pow` accepts an integer base and exponent and returns the base raised to that exponent.
  - Example: `pow(3, 4)` returns 81
  - Solve the problem recursively and without using loops.

An optimization

- Notice the following mathematical property:

$3^{12} = 531441 = 9^6$

$= (3^2)^6$

$531441 = (9^2)^3$

$= ((3^2)^2)^3$

  - When does this "trick" work?
  - How can we incorporate this optimization into our `pow` method?
  - What is the benefit of this trick if the method already works?

pow solution 2

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else if (exponent % 2 == 0) {
        // recursive case 1:  x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {
        // recursive case 2:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

Verification of Recursion

- How do we know a recursive method works?
  - Static analysis (static)
  - Testing (dynamic)
  - Formal verification (static)
- Formal Verification
  - Correctness proofs using principles learned in Discrete Math
    - Shows correctness of the algorithm on all valid inputs
  - What type of proof strategy is recursion similar to?

Verification of Recursion

- When proving an operation implementation is correct, the verifier has:
  - Preconditions: assume true at the beginning of the method (initial state(s))
  - Post Conditions (Ensures/Goal): what you must prove to be true at the end of the method (final state(s))

NC STATE UNIVERSITY

Verification of Recursion

- When proving a call of an operation is correct
  - Preconditions: prove that the input(s) to the operation is(are) correct before calling the operation
  - Post Conditions: assume operation's post conditions are true after the call

NC STATE UNIVERSITY

Method of Proof by Mathematical Induction
[Epp]

Consider a statement of the form, "For all integers, n>=a, a property P(n) is true." To prove such a statement, perform the following steps:

Step 1 (basis step): Show that the property is true for n=a.

Step 2 (inductive step): Show that for all integers m>=a, if the property is true for n=m then it is true for n=m+1. To perform this step,

suppose that the property is true for n=m, where m is any particular but arbitrarily chosen integer with m>=a. [*This supposition is called the* ***inductive hypothesis***.]

Then **show** that the property is true for n=m+1.

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
   if (j == 0) {
      return k;
   } else {
      int r = sum(j - 1, k);
      return r + 1;
   }
}
```


Example

[Based on code from RESOLVE Project]

For all integers j>=0, sum(j, k) = j+k.

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
   if (j == 0) {
      return k;
   } else {
      int r = sum(j - 1, k);
      return r + 1;
   }
}
```

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
   if (j == 0) {
      return k;
   } else {
      int r = sum(j - 1, k);
      return r + 1;
   }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 1 (basis step): Show that the property is true for j=0.

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
   if (j == 0) {
      return k;
   } else {
      int r = sum(j - 1, k);
      return r + 1;
   }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 1 (basis step): Show that the property is true for j=0.

Assume j=0. We must show that sum(0, k) = 0 +k.

Based on the code, sum(0, k) returns k.
k = 0 + k

Therefore true for j = 0.

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
    if (j == 0) {
        return k;
    } else {
        int r = sum(j - 1, k);
        return r + 1;
    }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 2 (inductive step): Show that for all integers m>=0, if the property is true for j=m then it is true for j=m+1.

suppose that the property is true for j=m, where j is any particular but arbitrarily chosen integer with m>=0.

Then **show** that the property is true for j=m+1. [sum(m+1, k) = (m+1) + k]

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
    if (j == 0) {
        return k;
    } else {
        int r = sum(j - 1, k);
        return r + 1;
    }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 2 (inductive step): Show that for all integers m>=0, if the property is true for j=m then it is true for j=m+1.

suppose that the property is true for j=m, where j is any particular but arbitrarily chosen integer with m>=0.
sum(m, k) = m+k

Then **show** that the property is true for j=m+1. [sum(m+1, k) = (m+1) + k]
sum(m+1, k) = sum(m, k) + 1

NC STATE UNIVERSITY

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
    if (j == 0) {
        return k;
    } else {
        int r = sum(j - 1, k);
        return r + 1;
    }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 2 (inductive step): Show that for all integers m>=0, if the property is true for j=m then it is true for j=m+1.

suppose that the property is true for j=m, where j is any particular but arbitrarily chosen integer with m>=0.
sum(m, k) = m+k

Then **show** that the property is true for j=m+1. [sum(m+1, k) = (m+1) + k]
sum(m+1, k) = sum(m, k) + 1

39

NC STATE UNIVERSITY

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
    if (j == 0) {
        return k;
    } else {
        int r = sum(j - 1, k);
        return r + 1;
    }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 2 (inductive step): Show that for all integers m>=0, if the property is true for j=m then it is true for j=m+1.

suppose that the property is true for j=m, where j is any particular but arbitrarily chosen integer with m>=0.
sum(m, k) = m+k

Then **show** that the property is true for j=m+1. [sum(m+1, k) = (m+1) + k]
sum(m+1, k) = sum(m, k) + 1
= (m+k) + 1 // based on
// assumption

CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty

40

Example

[Based on code from RESOLVE Project]

```
// Precondition: j >= 0
// Goal/Ensures:
//     result = j + k
int sum(int j, int k) {
    if (j == 0) {
        return k;
    } else {
        int r = sum(j - 1, k);
        return r + 1;
    }
}
```

For all integers j>=0, sum(j, k) = j+k.

Step 2 (inductive step): Show that for all integers m>=0, if the property is true for j=m then it is true for j=m+1.

suppose that the property is true for j=m, where j is any particular but arbitrarily chosen integer with m>=0.
sum(m, k) = m+k

Then **show** that the property is true for j=m+1. [sum(m+1, k) = (m+1) + k]
sum(m+1, k) = sum(m, k) + 1
= (m+k) + 1 // based on // assumption
= (m+1) + k

Therefore true for j>=0.

Exercise

- Write a recursive method `generateBinary` that accepts an integer and returns that number's representation in binary (base 2).
  - Example: `generateBinary (7)` returns 111
  - Example: `generateBinary (12)` returns 1100
  - Example: `generateBinary (42)` returns 101010

place	10	1
value	**4**	**2**

32	16	8	4	2	1
1	**0**	**1**	**0**	**1**	**0**

Slides are modified from those provided by Marty Stepp

```java
/**
 * Converts a decimal number to binary
 *
 * @param decimal number to convert
 * @return binary string of number
 * @throws IllegalArgumentException if decimal < 0
 */
public static String convertToBinary(int decimal) {
  if (decimal < 0) {
     throw new IllegalArgumentException("Negative value");
  }
  String binary = "";
  if (decimal == 0) { // Special case
     return "0";
  }
  while (decimal != 0) {
     binary = (decimal % 2) + binary;
     decimal /= 2;
  }
  return binary;
}
```


Case analysis

- Recursion is about solving a small piece of a large problem.
  - What is 69743 in binary?
    - Do we know *anything* about its representation in binary?
  - Case analysis:
    - What is/are easy numbers to print in binary?
    - Can we express a larger number in terms of a smaller number(s)?
  - Suppose we are examining some arbitrary integer `N`.
    - if `N`'s binary representation is `10010101011`
    - `(N / 2)`'s binary representation is `1001010101`
    - `(N % 2)`'s binary representation is `1`

Exercise

- Write a recursive method `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.
  - `isPalindrome("madam")` → `true`
  - `isPalindrome("racecar")` → `true`
  - `isPalindrome("step on no pets")` → `true`
  - `isPalindrome("able was I ere I saw elba")` → `true`
  - `isPalindrome("Java")` → `false`
  - `isPalindrome("rotater")` → `false`
  - `isPalindrome("byebye")` → `false`
  - `isPalindrome("notion")` → `false`

Exercise

- Write a method `crawl` accepts a `File` parameter and prints information about that file.
  - If the `File` object represents a normal file, just print its name.
  - If the `File` object represents a directory, print its name and information about every file/directory inside it, indented.

```
csc216
    handouts
        syllabus.doc
        lecture_schedule.xls
    homework
        1-sortedintlist
            ArrayIntList.java
            SortedIntList.java
            index.html
            style.css
```

  - **recursive data**: A directory can contain other directories.

`File` objects

- A `File` object (from the `java.io` package) represents a file or directory on the disk.

Constructor/method	Description
`File(`**String**`)`	creates `File` object representing file with given name
`canRead()`	returns whether file is able to be read
`delete()`	removes file from disk
`exists()`	whether this file exists on disk
`getName()`	returns file's name
`isDirectory()`	returns whether this object represents a directory
`length()`	returns number of bytes in file
`listFiles()`	returns a `File[]` representing files in this directory
`renameTo(`**File**`)`	changes name of file

Public/private pairs

- We cannot vary the indentation without an extra parameter:

```
private static void crawl(File f, String indent) {
```

- Often the parameters we need for our recursion do not match those the client will want to pass.

In these cases, we instead write a pair of methods:

1) a public, non-recursive one with the parameters the client wants: `public static void crawl(File f)`
2) a private, recursive one with the parameters we really need: `private static void crawl(File f, String indent)`

References

- Susanna S. Epp, *Discrete Mathematics with Applications*, Third Edition
- Material provided by Joseph Hollingsworth (IUS) and Murali Sitaraman (Clemson) as part of RESOLVE project