Part I.
Recursion as a Problem-Solving Technique

CS 200 Algorithms and Data Structures

Outline

- Backtracking
- Formal grammars
- Relationship between recursion and mathematical induction

Backtracking

- Problem solving technique that involves guesses at a solution.
- Retrace steps in reverse order and try new sequence of steps

The Eight Queens Problem

Place 8 Queens!
- No queen can attack any other queens.
Solution with recursion and backtracking

placeQueen (in currColumn:integer)
if ( currColumn > 8) {
The problem is solved
} else {
    while (unconsidered squares exist in currColumn and the problem is unsolved) {
        Determine if the next square is safe.
        if (such a square exists){
            place a queen in the square
            placeQueens(currColumn+1) // try next column
            if (no queen safe in currColumn+1) {
                remove queen from currColumn and try the next square in that col.
            }
        }
    }
}

Outline

• Backtracking
• Formal grammars
• Relationship between recursion and mathematical induction

Defining Languages

• Language: A set of strings of symbols from a finite alphabet.
• JavaPrograms = {strings w: w is a syntactically correct Java program}
• Grammar: the rules of a language
  – Determine whether a given string is in the language
  – Language Specifications

Some special symbols

• x|y means x or y
• x y means x followed by y
• <word> means any instance of word that the definition defines
Example

- Consider the language that the following grammar defines:
  - `<S> ::= % | <W> | %<S>`
  - `<W> ::= xy | x < W | y`
  - Write all strings that are in this language

Example: Java Identifier

- A grammar for the language
  - `JavaIds = {w: w is a legal Java identifier}`
- Java identifiers are the names of variables, methods, classes, packages and interfaces
  - Identifier: IdentifierChars but not a Keyword or BooleanLiterals or NullLiteral
  - IdentifierChars: JavaLetter or IdentifierChar or JavaLetterOrDigit
  - JavaLetter: any Unicode Character that is JavaLetter
  - JavaLetterOrDigit: any Unicode Character that is JavaLetterOrDigit
  - BooleanLiterals: any Unicode Character that is Boolean

A Grammar for the Java Identifier

- `<identifierChars> ::= <JavaLetter>|<identifierChars><JavaLetter>|<identifierChars><JavaDigit>|<identifierChars>|<letter>`
- `<letter> ::= a|b|...|z|A|B|...|Z`
- `<digit> ::= 0|1|...|9`
- An identifier is a letter, or an identifier followed by a letter, or an identifier followed by a digit.

Recognition of JavaId

```java
isId(in w: string): boolean
if (w is of length 1) {
  if (w is a letter or $ or _) {
    return true
  } else {
    return false
  }
} else if (the last character of w is a letter or a digit) {
  return isId(w minus its last character)
} else {
  return false
}
```

Example with "A2B"

- `isId("A2B")`
- `isId("A2")`
- `isId("A")`
- `true`
- `true`
- `true`

How to Define a Grammar for Palindromes

- A palindrome is a string that reads the same from left to right as it does from right to left.
- Palindromes = `{w: w reads the same left to right as right to left}`
Find a Rule to satisfy all the Palindromes

- Examples: RADAR, RACECAR, MADAM, [A nut for a jar of Tuna]
- If \( w \) is a palindrome
  - Then \( w \) minus its first and last characters is also a palindrome

Base cases

- Empty string is palindrome
- A string of length 1 is a palindrome

Grammar for the language Palindrome

- \(<pal> = \text{empty string} | <ch> | a <pal> a | b <pal> b | \ldots | Z <pal> Z \)
- \(<ch> = a | b | \ldots | z | A | B | \ldots | Z \)

Recursive Method for Recognizing Palindrome

```python
isPal(in w:string):boolean

if (w is an empty string or of length 1) {
    return true
} else if (w’s first and last characters are the same) {
    return isPal(w minus its first and last characters)
} else {
    return false
}
```

Example

- `isPal("RADAR")`
- `isPal("ADA")`
- `isPal("ADA")`
- `isPal("D")`
- `isPal("D")`
- TRUE
- TRUE
- TRUE

Algebraic Expressions

- Infix
  - Every binary operator appears between its operands
    \( a + b, a+(b*c), (a+b)*c \)
- Prefix
  - Operator appears before its operands
    \( +a, +a * b \)
- Postfix
  - Operator appears after its operands
    \( a * +b, a * b * c +, a + b * c \)
Examples

• \(- x \ 3 \ 8 \ + \ 6 \ 5\)
• \(+ \ -0.5 \ 2 \ \times \ 10 \ 2\)
• \(3 \ 8 \ \times \ 6 \ 5 \ + \ -\)
• \(5 \ 2 - 10 \ 2 \ \times \ +\)

Prefix Expressions

• \(<prefix> = <identifier> | <operator> <prefix> <prefix>\)
• \(<operator> = + | - | * | /\)
• \(<identifier> = a | b | \ldots | z\)

Recognize Prefix expressions

• Is the first character of input string an operator?
• Does the remainder of input string consist of two consecutive prefix expressions?

Recognize the end of prefix expressions

1: endPre (in first: integer, in second: integer): integer
2: if (first < 0 or first > last){return -1}// no prefix
3: ch = character at position first of strExp
4: if (ch is identifier){ return first }
5: else if ( ch is an operator) {
6: \ \ \ \ \ \ firstEnd = endPre(first +1, last)
7: \ \ \ \ \ \ if (firstEnd > -1) {
8: \ \ \ \ \ \ return endPre(firstEnd +1, last)
9: \ \ \ \ \ \ } else {
10: \ \ \ \ \ \ return -1
11: \ }
12: \ }else {
13: \ return -1
14: \ }

Example

• Trace of endPre (first, last), where strExp is \(+/ab-cd\)

Outline

• Backtracking
• Formal grammars
• Relationship between recursion and mathematical induction
Mathematical Induction in Dominos

- We have N dominos.
- If we push the 1st domino, will N dominos fall?
  - We should show:
    - If we push the 2nd one, it falls
    - For all of dominos, if the previous domino falls, next domino falls
- Process:
  - Show something works the first time
  - Assume that it works for this time
  - Show it will work for the next time, under the assumption
  - Conclusion, it works all the time

Mathematical Induction in Mathematics

- To prove that $P(n)$ is true for all positive integers $n$, where $P(n)$ is a propositional function,
- Two parts of mathematical induction
  - Basis step: verify that $P(1)$ is true
  - Inductive step: Show that the conditional statement $P(k) \rightarrow P(k+1)$ is true for all (positive, or non-negative) integers $k$.

Example

- Use mathematical induction to show that, $1+2+3+\ldots+n = n(n+1)/2$ for all positive integer $n$.

Recursion

- Specifies a solution to one or more base cases
- Then demonstrates how to derive the solution to a problem of an arbitrary size
  - From the smaller size of the same problem.

Mathematical Induction

- Proves a property about the natural numbers by
  - Proving the property about a base case and
  - Then proving that the property must be true for an arbitrary natural $N$ if it is true for the natural number smaller than $N$.
- Proving
  - (1) correctness of the recursive algorithm
  - (2) deriving the amount of recursive work it requires
Correctness of the Recursive Factorial Method

Definition of Factorial
factorial(n) = n (n-1) (n-2) ... 1 for any integer n > 0  
factorial(0) = 1

Definition of method fact(N)

1: fact (in n: integer): integer
2: if (n is 0) {
3:     return 1
4: } else {
5:     return n* fact(n-1)
6: }

Prove that the method fact computes the factorial of its arguments

Basis step:
fact(0) = 1

Inductive Step:
Show that for an arbitrary positive integer k, if fact(k) returns k!, fact(k+1) returns (k+1)!
Assume that, fact(k) = k (k-1) (k-2) ... 2 1  
For n = k+1,  
Show that fact(k+1) returns (k+1) k (k-1) (k-2) ... 2 1

The Towers of Hanoi

• Only one disk may be moved at a time.
• No disk may be placed on top of a smaller disk.

States in the Towers of Hanoi

SolveTower(3,A,B,C)
SolveTower(2,A,C,B)
SolveTower(1,A,B,C)
SolveTower(2,C,B,A)
SolveTower(1,A,B,C)
SolveTower(1,A,C,B)
SolveTower(1,B,C,A)
SolveTower(1,C,A,B)
SolveTower(1,C,B,A)
SolveTower(1,A,B,C)

Recursive Solution

if (count is 1) {
    Move a disk directly from source to destination
} else{
    solveTowers(count-1, source, spare, destination)
    solveTowers(1, source, destination, spare)
    solveTowers(count-1, spare, destination, source)
}
## Cost of Towers of Hanoi

- If we have $N$ disks, how many moves does `solveTowers()` make to solve the problem?
- From the software
  
  
  \[
  \begin{align*}
  \text{move}(1) &= 1 \\
  \text{move}(N) &= \text{move}(N-1) + 1 + \text{move}(N-1) \quad (\text{if } N > 1)
  \end{align*}
  \]
- A closed form formula for the number of moves that `solveTowers` requires for $N$ disks:
  
  \[
  \text{move}(N) = 2^N - 1 \quad (\text{for all } N \geq 1)
  \]
- Is this true for the `solveTowers()` method with $N$ disks?

## Proof

- **Basis Step**
  
  Show that the property is true for $N = 1$.
  
  \[
  2^1 - 1 = 1,
  \]
  
  which is consistent with the recurrence relation's specification that \( \text{move}(1) = 1 \)

- **Inductive Step**
  
  Property is true for an arbitrary $k$ \( \Rightarrow \) property is true for $k+1$

  Assume that the property is true for $N = k$

  \[
  \text{move}(k) = 2^k - 1
  \]

  Show that the property is true for $N = k + 1$

## Proof – cont.

- \( \text{move}(k+1) = 2 \times \text{move}(k) + 1 \)

  \[
  = 2 \times (2^k - 1) + 1
  = 2^{k+1} - 1
  \]

  Therefore the inductive proof is complete.

## Readings for next class

- Chap. 9  Advanced Java Topics from Prichard