Algorithmic Design

- The “science” of problem solving
- Using invariants to reason about problem solving
- Design Patterns
- Representing problems graphically

Algorithms

- What is Computer Science?
  
  What is it that distinguishes it from the separate subjects with which it is related? What is the linking thread which gathers these disparate branches into a single discipline? My answer to these questions is simple — *it is the art of programming a computer*. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex.
  
  C.A.R. (Tony) Hoare

- An *algorithm* is a recipe, a plan, or a set of instructions
- What we think about and what we teach is often how to design algorithms or just solve problems.

Problem Solving

- Some believe that being a good programmer will be a prerequisite for being a good mathematician
  - Computer-aided proofs are big (four color theorem, Kepler’s conjecture)
  - Computer programs are very formally complete and precise
- Teachers often speak of a magical “problem solving intuition”
- Does such a thing exist?
- Is it really just experience and pattern recognition?
- What are some tools to help learning programmers to solve problems?

Loop Invariants

- Want to reason about the correctness of a proposed iterative solution
- Loop invariants provide a means to effectively about the correctness of code

while !done do
  // what is true at every step
  // Update/iterate
  // maintain invariant
  od
Bean Can game

- Can contains $N$ black beans and $M$ white beans initially
- Emptied according the following repeated process
  - Select two beans from the can
  - If the beans are:
    - same color: put a black bean back in the can
    - different colors: put a white bean back in the can
  - Player who chooses the color of the remaining bean wins the game
- Analyze the link between the initial state and the final state
- Identify a property that is preserved as beans are removed from the can
  - Invariant that characterizes the removal process

Bean Can Algorithm

```
while (num-beans-in-can > 1) do
    pick 2 beans randomly
    if bean1-color == bean2-color then
        put-back black bean
    else
        put-back white bean
    od
```

Bean Can Analysis

- What happens each turn?
  - Number of beans in can is decreased by one
  - Number of white beans is either reduced by 2 or 0
  - Number of black beans is either reduced by 1 or 0
- Examine the final states for 2 bean and 3 bean initial states
- Any guesses for the correct strategy?
- What is the process invariant?

The Game of Nim

- Two Piles of counters with $N$ and $M$ counters in each pile
- 2 players take turns:
  - Remove some number of counters ($\geq 1$) from one pile
  - Player who removes last counter wins
- Properties
  - Complete information: could exhaustively search for winning solution
  - Impartial: same moves are available for each player
**Nim Analysis**

- Denote state by (x,y): number of counters in each pile
- What about simple case of (1,1)?
- For whom is (1,1) a “safe” state?
- How about (1,2) or (1,3)?
- How about (2,2)?
- What is the invariant to be preserved by the winning player?

**Nim Algorithm**

```plaintext
// reach a state (x,y) where x=y on opponent’s turn and then follow below algorithm

while !empty(pile1) && !empty(pile2) do
  let opponent remove q counters from a pile
  remove q counters from other pile
od
```

**City Battle**

- Robots placed in opposite corners of rectangular city (nxm)
- City map is grid of horizontal and vertical streets
- Robots move in alternating turns, moving either horizontally or vertically
- The goal of each robot is to have its opponent enter its line of fire (vertically or horizontally)
- What is the strategy for winning the game?
  - Hint: Another Loop invariant

**Dropping Glass Balls**

- Tower with N Floors
- Given 2 glass balls
- Want to determine the lowest floor from which a ball can be dropped and will break
- How?
  - What is the most efficient algorithm?
  - How many drops will it take for such an algorithm (as a function of N)?
Numbers from Ends

- Game begins with some even number of numbers on a line
  
  10  5  7  9  6  12

- Players take turns removing numbers from the ends while keeping running sum of numbers collected so far

- Player with largest sum wins

- Complete information but how to win without search?

Pennies Heads Up

From Car Talk!

- You're sitting at a table with a bunch of pennies on it. Some are facing heads up and some are facing tails up. You're wearing a blindfold, and you're wearing mittens so that you can't actually feel the coins and tell which side is facing up.

- I will tell you that a certain number of the pennies are facing heads up. Let's say 10 are facing heads up.

- Is it possible to separate those pennies into two groups, so that each group has the same number of pennies facing heads up? How do you do it?

  ➣ Pennies can be flipped or moved as much as needed

Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

- Alexander et. al, 1977

- A text on architecture!

- What is a programming or design pattern?

- Why are patterns important?

What is a pattern?

- "... a three part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, ..., and the rule which tells us how to create that thing, and when we must create it."

  Christopher Alexander

  ➣ name
  ➣ problem
  ➣ solution
  ➣ consequences

- name factory, aka virtual constructor

- problem delegate creation responsibility: expression tree nodes

- solution createFoo() method returns aFoo, bFoo,...

- consequences potentially lots of subclassing, ...

- more a recipe than a plan, micro-architecture, frameworks, language idioms made abstract, less than a principle but more than a heuristic

- patterns capture important practice in a form that makes the practice accessible
Patterns are discovered, not invented

- You encounter the same “pattern” in developing solutions to programming or design problems
  - develop the pattern into an appropriate form that makes it accessible to others
  - fit the pattern into a language of other, related patterns
- Patterns transcend programming languages, but not (always) programming paradigms
  - OO folk started the patterns movement
  - language idioms, programming templates, programming patterns, case studies

Programming Problems

- Microsoft interview question (1998)
- Dutch National Flag problem (1976)
- Remove Zeros (AP 1987)
- Quicksort partition (1961, 1986)
- Run-length encoding (SIGCSE 1998)

Removing Duplicates

```cpp
void crunch(tvector<string> list)
{
    int lastUniqueIndex = 0;
    string lastUnique = list[0];
    for(int k=1; k < list.size(); k++)
    {
        string current = list[k];
        if (current != lastUnique)
        {
            list[++lastUniqueIndex] = current;
            lastUnique = current;
        }
    }
    list.resize(lastUniqueIndex+1);
}
```

Solving (related) problems

- Sometimes it is not clear that problems are related, or how problems are related
- Educators sometimes do not know what they know, so cannot convey knowledge of how to solve problems to students
  - often students don’t see programming problems as related, or see them related by language features rather than by higher-level features/dependencies
  - it’s often difficult for students to appreciate why one method of solving a problem is better without a context to see the solution in force more than once
- Using patterns can help make knowledge gleaned over many years accessible to those new to the field
  - patterns may be useful in connecting problems and providing a framework for categorizing solutions
One loop for linear structures

- Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but data is stored sequentially, e.g., in an array or file. Programming based on control leads to more problems than programming based on structure. Therefore, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately guarded conditionals to implement the control.

Consequences: one loop really means loop according to structure, do not add loops for control: what does the code look like for run-length encoding example? What about efficiency?

Coding Pattern

- Name:
  - one loop for linear structures

- Problem:
  - Sequential data, e.g., in an array or a file, must be processed to perform some algorithmic task. At first it may seem that multiple (nested) loops are needed, but developing such loops correctly is often hard in practice.

- Solution:
  - Let the structure of the data guide the coding solution. Use one loop with guarded/if statements when processing one-dimensional, linear/sequential data

- Consequences:
  - Code is simpler to reason about, facilitates development of loop invariants, possibly leads to (slightly?) less efficient code

Observer/Observable

- When the creatures move, the world should show their movement
  - when a program executes, the program view changes
  - each observable (creature) notifies its observers (world listener, program listener) when observable changes
  - separate the model from the view, especially useful when developing GUI programs, allows multiple views of the same model

Iterator

- Collections must support access to their elements, use a separate class for access, e.g., supporting forward, backward, sequential, random access
  - iterators are essential in STL, Enumeration used in Java 1.1, Iterator used in 1.2

- Use the Iterator pattern early in curriculum to hide platform-specific code (e.g., in C++), to hide complicated code (e.g., I/O in Java), and to introduce an important pattern
  - WordStreamIterator in C++ and in Java
  - Directory reading/processing classes in both languages

- Internal iterators useful also, even in non OO languages
Design patterns you shouldn’t miss

- **Iterator**
  - useful in many contexts, see previous examples, integral to both C++ and Java
- **Factory**
  - essential for developing OO programs/classes, e.g., create iterator from a Java 1.2 List? `list.iterator()`
- **Composite**
  - essential in GUI/Widget programming, widgets contain collections of other widgets
- **Adapter/Façade**
  - replug-and-play, hide details
- **Observer/Observable, Publish/Subscribe, MVC**
  - separate the model from the view, smart updates

*(Fox - Rooster - Corn) River*

- Your goal is to cross a river with cargo
  - Fox (F)
  - Rooster (R)
  - Corn (C)
- You have a canoe that can only hold 1 item
- Fox eats Rooster if they’re left alone
- Rooster eats Corn if they’re left alone
- But, just to make it interesting...
  - Associate cost with each object
    - Fox (F<sub>c</sub>)
    - Rooster (R<sub>c</sub>)
    - Corn (C<sub>c</sub>)
  - Find optimal sequence of moves to minimize cost given all possible values for F<sub>c</sub>, R<sub>c</sub>, C<sub>c</sub>
  - Return sequence and cost

One key insight

- Decouple cost function & search
  - First focus on the search
  - Then, look at the cost calculation

Naïve Approach

- Brute-force British-museum search
- This is what most do in their head
- Often folks get stuck
  - Deep, deep, deep down in the search tree, they’ve fallen and can’t get back up
  - Perhaps there’s a better way to visualize?
Leverage Proprieception

- Proprieception:
  - Close your eyes and ask yourself:
    - Am I lying down, standing up or sitting down?
    - Are my hands to my side or crossed in front?
  - *Innate awareness of self positioning*

- Body-syntonic approach
  - Imagine that you are the principal player
  - *Perhaps there's a better way to visualize?*

Representation

- Each of F, R, C is either here or there
  - If one of these is here, underline (e.g., F)
  - If one of these is there, normal (e.g., F)
- We can consider each location a bit
  - 3 bits total
  - 8 states (e.g., FRC)
- Insight:
  - Map these to the vertices of a cube! [Gardner]
Design 33  
**Rooster**  
Representation (graphically)  

Design 34  
**Corn**  
Representation (graphically)  

( Fox - Rooster - Corn ) River  
Representation (graphically)  

Initial & final states
Show valid canoe nodes

- Rooster lives!
- Some stable

Walk from start to finish
- Alternate steps
If we alternate steps, remove edges where can’t

If we alternate steps, remove edges where can’t

If we alternate steps, remove edges where can’t

We’re left with a graph which highlights the 2 solutions!
We're left with a graph which highlights the 2 solutions!

Insight: Cost functions are equal!

Cost = $R_c + F_c + R_c + C_c + R_c$
How this relates to CS

- Proprieception & Body-syntonic problem
- Importance of understanding the problem before trying to brute-force it
- Importance of a good representation
- Introduces and reinforces importance of Gray coding (1 bit changes per state)
- Problem Decomposition, w/ and w/out cost
- Nice, small, gentle search-space

Conclusions

- Problem solving/algorithmic design is key part of computer science
- Games and puzzles are be useful pedagogical techniques
- Practice makes “intuition”
- Reasoning about loop invariants helps one reason about code
- Recognize and utilize patterns

References

- Games
  - Berlekamp, Conway, Guy
    - “Winning Ways” (vols I and II)
    - “Fair Game” [Guy]
- Puzzles
  - Martin Gardner’s many books
    - “Aha Gotcha”
    - “Aha Insight”