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?

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 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 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);
}
```

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

Analysis: Algorithms and Data Structures

- We need a vocabulary to discuss performance and to reason about alternative algorithms and implementations
  - What’s the best way to sort, why?
  - What’s the best way to search, why?
- We need both empirical tests and analytical/mathematical reasoning
  - Given two methods, which is better? Run them to check.
    - 30 seconds vs. 3 seconds, easy. 5 hours vs. 2 minutes, harder
  - Which is better? Analyze them.
    - Use mathematics to analyze the algorithm, the implementation is another matter

Multiplying and adding big-Oh

- Suppose we do a linear search then we do another one
  - What is the complexity?
  - If we do 100 linear searches?
  - If we do n searches on a vector of size n?
- What if we do binary search followed by linear search?
  - What are big-Oh complexities? Sum?
  - What about 50 binary searches? What about n searches?
- What is the number of elements in the list (1,2,2,3,3,3)?
  - What about (1,2,2, …, n,n,…,n)?
  - How can we reason about this?

Helpful formulae

- We always mean base 2 unless otherwise stated
  - What is log(1024)?
  - $\log(xy) = \log(x) + \log(y)$
  - $y \log(x)$
  - $n \log(2) = n$
  - $2^{(\log n)} = n$
  - $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 = \sum_{i=0}^{n} 2^i$
  - $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \sum_{i=1}^{1} i$
  - $a + ar + ar^2 + \ldots + ar^{n-1} = a(r^n - 1)/(r-1) = \sum_{i=0}^{n-1} ar^i$
Different measures of complexity

- **Worst case**
  - Gives a good upper-bound on behavior
  - Never get worse than this
  - Drawbacks?

- **Average case**
  - What does average mean?
  - Averaged over all inputs? Assuming uniformly distributed random data?
  - Drawbacks?

- **Best case**
  - Linear search, useful?

Recurrences

- **Counting nodes**
  ```
  int length(Node * list)
  {
    if (0 == list) return 0;
    else return 1 + length(list->next);
  }
  ```

- **What is complexity? justification?**
  - $T(n)$ = time to compute length for an n-node list
    - $T(n) = T(n-1) + 1$
    - $T(0) = 1$

- instead of 1, use $O(1)$ for constant time
  - independent of $n$, the measure of problem size

Solving recurrence relations

- **plug, simplify, reduce, guess, verify?**

  - $T(n) = T(n-1) + 1$
  - $T(0) = 1$

  - $T(n-1) = T(n-1-1) + 1$
  - $T(n) = [T(n-2) + 1] + 1 = T(n-2)+2$
  - $T(n-2) = T(n-2-1) + 1$
  - $T(n) = [(T(n-3) + 1) + 1] + 1 = T(n-3)+3$

  - $T(n) = T(n-k) + k$ **find the pattern!**
  - Now, let $k=n$, then $T(n) = T(0)+n = 1+n$

- **get to base case, solve the recurrence: $O(n)$**

Consider merge sort for linked lists

- **Given a linked list, we want to sort it**
  - Divide the list into two equal halves
  - Sort the halves
  - Merge the sorted halves together

- **What’s complexity of dividing an n-node in half?**
  - How do we do this?

- **What’s complexity of merging (zipping) two sorted lists?**
  - How do we do this?

- **$T(n) = time\ to\ sort\ n-node\ list = 2\ T(n/2) + O(n)$** why?
### sidebar: solving recurrence

\[
T(n) = 2T(n/2) + n \\
T(1) = 1
\]

(Note: \(n/2/2 = n/4\))

\[
T(n) = 2[2T(n/4) + n/2] + n \\
= 4 T(n/4) + n + n \\
= 8T(n/8) + 3n \\
= \ldots \text{eureka!} \\
= 2^k T(n/2^k) + kn
\]

Let \(2^k = n\)

\[
k = \log n, \text{ this yields } 2^{\log n} T(n/2^{\log n}) + n(\log n) \\
= n T(1) + n(\log n) \\
\leq O(n \log n)
\]

### Complexity Practice

- **What is complexity of Build? (what does it do?)**

  ```java
  Node * Build(int n) 
  {
    if (0 == n) return 0;
    Node * first = new Node(n,Build(n-1));
    for(int k = 0; k < n-1; k++) {
      first = new Node(n,first);
    }
    return first;
  }
  ```

- **Write an expression for \(T(n)\) and for \(T(0)\), solve.**

### Recognizing Recurrences

- **Solve once, re-use in new contexts**
  - \(T\) must be explicitly identified
  - \(n\) must be some measure of size of input/parameter
    - \(T(n)\) is the time for quicksort to run on an \(n\)-element vector

  \[
  T(n) = T(n/2) + O(1) \quad \text{binary search} \quad O(\log n) \\
  T(n) = T(n-1) + O(1) \quad \text{sequential search} \quad O(n) \\
  T(n) = 2T(n/2) + O(1) \quad \text{tree traversal} \quad O(n) \\
  T(n) = 2T(n/2) + O(n) \quad \text{quicksort} \quad O(n \log n) \\
  T(n) = T(n-1) + O(n) \quad \text{selection sort} \quad O(n^2)
  \]

- **Remember the algorithm, re-derive complexity**

### Stack: What problems does it solve?

- **Stacks are used to avoid recursion, a stack can replace the implicit/actual stack of functions called recursively**
- **Stacks are used to evaluate arithmetic expressions, to implement compilers, to implement interpreters**
  - The Java Virtual Machine (JVM) is a stack-based machine
  - Postscript is a stack-based language
  - Stacks are used to evaluate arithmetic expressions in many languages

- **Small set of operations: LIFO or last in is first out access**
  - Operations: push, pop, top, create, clear, size
  - More in postscript, e.g., swap, dup, rotate, ...
Simple stack example

- **tstack** is a templated class, stores any type of value that can be assigned (like *tvector*).
  - Implemented simply using a vector, what does pop do?

```cpp
tstack<int> s;
s.push(2);
s.push(3);
s.push(1);
cout << s.size() << endl;
cout << s.top() << endl;
s.pop();
cout << s.top() << endl;
int val;
s.pop(val);
cout << val << endl;
```

Templated class, .h ok, .cpp ugly

- See *tstack.h* for example

```cpp
template <class Type>
class tstack
{
  public:
  tstack( ) ;                   // construct empty stack
  const Type & top( ) const;   // return top element
  bool isEmpty( ) const;      // return true iff empty
  int size( ) const;           // # elements

  void push( const Type & item ); // push item

  But look at part of *stack.cpp*, class is templated (ugly?)

template <class Type>
bool tstack<Type>::isEmpty() const
{
  return myElements.size() == 0;
}
```

Postfix, prefix, and infix notation

- Postfix notation used in some HP calculators
  - No parentheses needed, precedence rules still respected
  - Read expression
    - For number/operand: push
    - For operator: pop, pop, operate, push

- See *postfix.cpp* for example code, key ideas:
  - Read character by character, check state of expression
  - Note: putback character on stream, only last one read
- What about prefix and infix notations, advantages?

Prefix notation in action

- Scheme/LISP and other functional languages tend to use a prefix notation

```Scheme
(define (square x) (* x x))
(define (expt b n)
  (if (= n 0)
    1
    (* b (expt b (- n 1))))
```
**Postfix notation in action**

- Practical example of use of stack abstraction
- Put operator after operands in expression
  - Use stack to evaluate
    - operand: push onto stack
    - operator: pop operands push result
- PostScript is a stack language mostly used for printing
  - drawing an X with two equivalent sets of code

  ```
  %!
  200 200 moveto
  100 100 rlineto
  200 300 moveto
  100 -100 rlineto
  stroke showpage
  %!
  ```

**Queue: another linear ADT**

- FIFO: first in, first out, used in many applications
  - Scheduling jobs/processes on a computer
  - Tenting policy?
  - Computer simulations
- Common operations (as used in `tqueue.h/tqueue.cpp`)
  - Add to back, remove from front
    - Called enqueue, dequeue, like `s.push()` and `s.pop()`
    - Analog of `top()` is `front()`
- Also used in level-order tree traversal, similar to pre-order without recursion but using stack
  - See code in `treelevel.cpp`

**Stack and Queue implementations**

- Different implementations of queue (and stack) aren’t really interesting from an algorithmic standpoint
  - Complexity is the same, performance may change (why?)
  - Use vector or linked list, any sequential structure

- Linked list is easy for stack, where to add/remove nodes?

- Linked list is easy for queue, where to add/remove nodes?
  - Use circular linked list, why?

- Vector for queue is tricky, need ring buffer implementation, add but wrap-around if possible before growing
  - Tricky to get right (see `tqueue.h, tqueue.cpp`)

**Using linear data structures**

- We’ve studied vectors, stacks, queues, which to use?
  - It depends on the application
  - Vector is multipurpose, why not always use it?
    - Make it clear to programmer what’s being done
    - Other reasons?

- Other linear ADTs exist
  - List: add-to-front, add-to-back, insert anywhere, iterate
    - Alternative: create, head, tail (see `CList<>` in tapestry)
    - Linked-list nodes are concrete implementation
  - Deque: add-to-front, add-to-back, random access
    - Why is this “better” than a vector?
    - How to implement?