What’s easy, hard, first?

- Always do the hard part first. If the hard part is impossible, why waste time on the easy part? Once the hard part is done, you’re home free.

- Always do the easy part first. What you think at first is the easy part often turns out to be the hard part. Once the easy part is done, you can concentrate all your efforts on the hard part.

- Be sure to test whatever you do in isolation from other changes. Minimize writing lots of code then compiling/running/testing.
  - Build programs by adding code to something that works
  - How do you know it works? Only by testing it.

Program tips 7.3, 7.4

Designing and Implementing a class

- Consider a simple class to implement sets (no duplicates) or multisets (duplicates allowed)
  - Initially we’ll store strings, but we’d like to store anything, how is this done in C++?
  - How do we design: behavior or implementation first?
  - How do we test/implement, one method at a time or all at once?

- Tapestry book shows string set implemented with singly linked list and header node
  - Templated set class implemented after string set done
  - Detailed discussion of implementation provided
- We’ll look at a doubly-linked list implementation of a MultiSet

Interfaces and Inheritance

- Programming to a common interface is a good idea
  - I have an iterator named FooIterator, how is it used?
    - Convention enforces function names, not the compiler
  - I have an ostream object named out, how is it used?
    - Inheritance enforces function names, compiler checks

- Design a good interface and write client programs to use it
  - Change the implementation without changing client code
  - Example: read a stream, what’s the source of the stream?
    - file, cin, string, web, ...

- C++ inheritance syntax is cumbersome, but the idea is simple:
  - Design an interface and make classes use the interface
  - Client code knows interface only, not implementation

Iterators and alternatives

LinkedMultiSet ms;
ms.add("bad");
ms.add("good");
ms.add("ugly");
ms.add("good");
ms.print();

- What will we see in the implementation of print()? How does this compare to total()? What about the iterator code below?

MultiSetIterator it(ms);
for(it.Init(); it.HasMore(); it.Next()){
cout << it.Current() << endl;
}
Iterators and alternatives, continued

- In current MultiSet class every new operation on a set (total, print, union, intersection, …) requires a new member function/method
  - Change .h file, client code must be recompiled
  - Public interface grows unwieldy, minimize # of methods

- Iterators require class tightly coupled to container class
  - Friend class often required, const problems but surmountable
  - Harder to implement, but most flexible for client programs

- Alternative, pass function into MultiSet, function is applied to every element of the set
  - How can we pass a function? (Think AnaLenComp)
  - What’s potential difference compared to Iterator class?

Using a common interface

- We’ll use a function named applyOne(…)  
  - It will be called for each element in a MultiSet
    - An element is a (string, count) pair
  - Encapsulated in a class with other behavior/state

```cpp
void Printer::applyOne(const string& s, int count)
{
    cout << count << "\t" << s << endl;
}

void Total::applyOne(const string& s, int count)
{
    myTotal += count;
}
```

What is a function object?

- Encapsulate a function in a class, enforce interface using inheritance or templates
  - Class has state, functions don’t (really)
  - Sorting using different comparison criteria as in extra-credit for Anagram assignment

- In C++ it’s possible to pass a function, actually use pointer to function
  - Syntax is awkward and ugly
  - Functions can’t have state accessible outside the function (how would we count elements in a set, for example)?
  - Limited since return type and parameters fixed, in classes can add additional member functions

Why inheritance?

- Add new shapes easily without changing code
  - Shape * sp = new Circle();
  - Shape * sp2 = new Square();

- abstract base class:
  - interface or abstraction
  - pure virtual function

- concrete subclass
  - implementation
  - provide a version of all pure functions

- “is-a” view of inheritance
  - Substitutable for, usable in all cases as-a

User’s eye view: think and program with abstractions, realize different, but conforming implementations
Guidelines for using inheritance

- Create a base/super/parent class that specifies the behavior that will be implemented in subclasses
  ➤ Functions in base class should be virtual
    • Often pure virtual (= 0 syntax), interface only
  ➤ Subclasses do not need to specify virtual, but good idea
    • May subclass further, show programmer what’s going on
  ➤ Subclasses specify inheritance using: public Base
    • C++ has other kinds of inheritance, stay away from these
  ➤ Must have virtual destructor in base class

- Inheritance models “is-a” relationship, a subclass is-a parent-class, can be used-as-a, is substitutable-for
  ➤ Standard examples include animals and shapes

Inheritance guidelines/examples

- Virtual function binding is determined at run-time
  ➤ Non-virtual function binding (which one is called) determined at compile time
  ➤ Need compile-time, or late, or polymorphic binding
  ➤ Small overhead for using virtual functions in terms of speed, design flexibility replaces need for speed
    • Contrast Java, all functions “virtual” by default
- In a base class, make all functions virtual
  ➤ Allow design flexibility, if you need speed you’re wrong, or do it later
- In C++, inheritance works only through pointer or reference
  ➤ If a copy is made, all bets are off, need the “real” object

See students.cpp, school.cpp

- Base class student doesn’t have all functions virtual
  ➤ What happens if subclass uses new name() function?
    • name() bound at compile time, no change observed

- How do subclass objects call parent class code?
  ➤ Use class::function syntax, must know name of parent class

- Why is data protected rather than private?
  ➤ Must be accessed directly in subclasses, why?
  ➤ Not ideal, try to avoid state in base/parent class: trouble
    • What if derived class doesn’t need data?

Wordcounting, sets, and the web

- How does Google store all web pages that include a reference to “peanut butter with mustard”
  ➤ What efficiency issues exist?
  ➤ Why is Google different (better)?

- How do readwords.cpp and readwordslist.cpp differ?
  ➤ Mechanisms for insertion and search
  ➤ How do we discuss? How do we compare performance?

- If we stick with linked lists, how can we improve search?
  ➤ Where do we want to find things?
  ➤ What do thumb indexes do in a dictionary?
    • What about 256 different linked lists?
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?
  - How do we discuss these questions?

- We need both empirical tests and analytical/mathematical reasoning
  - Given two methods, which is better? Run them.
  - 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

Empirical and Theoretical results

- We run the program to time it (e.g., using CTimer)
  - What do we use as data? Why does this matter
  - What do we do about the kind of machine being used?
  - What about the load on the machine?

- Use mathematical/theoretical reasoning, e.g., sequential search
  - The algorithm takes time linear in the size of the vector
  - Double size of vector means double the time
  - What about searching twice, is this still linear?

- We use big-Oh, a mathematical notation for describing a class of functions

Toward O-notation (big-Oh)

- We investigated two recursive reverse-a-linked-list function (see revlist.cpp). The cost model was “->next dereferences”
  - Cost(n) = 2n + 1 + Cost(n-1) for loop function
  - Cost(n) = 5 + Cost(n-1) non-loop
  - How do these compare when n = 1000? What about a closed-form solution rather than a recurrence relation?

- What about
  - C(n) = n^2 + 2n - 2
  - C(n) = 5n - 4

- How do we derive these?

<table>
<thead>
<tr>
<th>n</th>
<th>Loop Cost</th>
<th>Non-loop Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>100</td>
<td>10,198</td>
<td>496</td>
</tr>
<tr>
<td>1000</td>
<td>1,001,198</td>
<td>4,996</td>
</tr>
</tbody>
</table>

Asymptotic Analysis

- We want to compare algorithms and methods. We care more about what happens when the input size doubles than about what the performance is for a specific input size
  - How fast is binary search? Sequential search?
  - If vector doubles, what happens to running time?
  - We analyze in the limit, or for “large” input sizes

- But we do care about performance: what happens if we search once vs. one thousand times?
  - What about machine speed: 300 Mhz vs 1Ghz?
  - How fast will machines get? Moore’s law

- We also care about space, but we’ll first talk about time
What is big-Oh about?

- Intuition: avoid details when they don’t matter, and they don’t matter when input size \( N \) is big enough
  - For polynomials, use only leading term, ignore coefficients
    
    \[
    y = 3x \\
    y = 6x - 2 \\
    y = 15x + 44 \\
    y = x^2 \\
    y = x^2 - 6x + 9 \\
    y = 3x^2 + 4x
    \]

- The first family is \( O(n) \), the second is \( O(n^2) \)
  - Intuition: family of curves, generally the same shape
  - More formally: \( O(f(n)) \) is an upper-bound, when \( n \) is large enough the expression \( c \cdot f(n) \) is larger
  - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time

More on O-notation, big-Oh

- Running time for input of size \( N \) is 10\( N \) or 2\( N \) or \( N + 1 \)
  - Running time is \( O(N) \), call this linear
  - Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
  - Allows us to compare algorithms in the limit
    - \( 20N \) hours vs \( N^2 \) microseconds

- O-notation is an upper-bound, this means that \( N \) is \( O(N) \), but it is also \( O(N^2) \) however, we try to provide tight bounds
  - What is complexity of binary search:
  - What is complexity of linear search:
  - What is complexity of selection sort:
  - Bubble Sort, Quick sort, other sorts?

Running times @ \( 10^6 \) instructions/sec

<table>
<thead>
<tr>
<th>( N )</th>
<th>( O(\log N) )</th>
<th>( O(N) )</th>
<th>( O(N \log N) )</th>
<th>( O(N^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000064</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.00100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>

Determining complexity with big-Oh

- runtime, space complexity refers to mathematical notation for algorithm (not really to code, but ok)
- typical examples:
  ```
  sum = 0;
  for(k=0; k < n; k++)
    { 
      if (a[k] == key) sum++;
      min = k;
    }
  return sum;
  
  for(j=k+1; j < n; j++)
    if (a[j] < a[min]) min = j;
  Swap(a[min],a[k]);
  ```
- what are complexities of these?
**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)?
  - $\log(xy) = \log(x) + \log(y)$
  - $\log(2^n) = 2\log n$

- Sums (also, use sigma notation when possible)
  - $1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1$
  - $1 + 2 + 3 + \ldots + n = n(n+1)/2$
  - $a + ar + ar^2 + \ldots + ar^{n-1} = a(r^n - 1)/(r-1)$

**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**
  ```c
  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?

\[
T(n) = T(n-1) + 1 \\
T(0) = 1
\]

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

find the pattern!

- get to base case, solve the recurrence

Why we study recurrences/complexity?

- Tools to analyze algorithms
  - Machine-independent measuring methods
  - Familiarity with good data structures/algorithms

- What is CS person: programmer, scientist, engineer?
  scientists build to learn, engineers learn to build

- Mathematics is a notation that helps in thinking, discussion, programming

Complexity Practice

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

```cpp
Node * Build(int n)  
{  
  if (0 == n) return 0;  
  else  
    {  
      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

\[
T(n) = T(n/2) + \mathcal{O}(1) \quad \text{binary search} \quad \mathcal{O}(\quad)
\]

\[
T(n) = T(n-1) + \mathcal{O}(1) \quad \text{sequential search} \quad \mathcal{O}(\quad)
\]

\[
T(n) = 2T(n/2) + \mathcal{O}(1) \quad \text{tree traversal} \quad \mathcal{O}(\quad)
\]

\[
T(n) = 2T(n/2) + \mathcal{O}(n) \quad \text{quicksort} \quad \mathcal{O}(\quad)
\]

\[
T(n) = T(n-1) + \mathcal{O}(n) \quad \text{selection sort} \quad \mathcal{O}(\quad)
\]

- Remember the algorithm, re-derive complexity