Is it Computer Science, Software Engineering, or Hacking?

Dr. Bruce K. Haddon
The Origins of the Problem

- A parser processing a Unicode input file, with a need to build *first* and *follow* sets (traditionally done with bit sets);

- Unicode has code points in the range $0 \sim 2^{21}$

- The Java JDK has a class, **BitSet**: range $0 \sim 2^{32} - 1$

- A match made in heaven! ?
Enter Reality!

- To store a single bit using `BitSet` at bit $2^{32}-1$ takes $2^{27}$ 32-bit words ($2^{26}$ 64bit “words”), not counting any Java object overhead.

- The amount of memory available in a 32-bit JVM in a 4 GB Windows system is approximately $2^{26}$ 32-bit words at best (not quite $2^{27}$ on a Linux system).

**Oops!**
A sparse bitset is a large collection of boolean values where many of the values are off (or false). For maintaining these sparsely populated sets, the BitSet class can be very inefficient. Since the majority of the bits will be off, space will be occupied to store nothing. For working with these sparse bitsets, you can create an alternate representation, backed instead by a hashtable, or HashMap. Only those positions where a value is set are then stored in the mapping.
The Instructions

To create a sparse bitset, subclass **BitSet** and override the necessary methods (everything). The skeleton code should help get your started, so you can focus on the set-oriented routines. The following UML diagram shows you the **BitSet** operations:

**Skeleton Code**

*SparseBitSet.java*

*Tester.java*

**Tasks**
**BitSet (UML)**

- `+BitSet()`
- `+BitSet(int : nbits)`
- `+and(set : BitSet) : void`
- `+andNot(set : BitSet) : void`
- `+cardinality() : int`
- `+clear() : void`
- `+clear(i : int) : void`
- `+clear(i : int, j : int) : void`
- `+clone() : Object`
- `+equals(obj : Object) : boolean`
- `+flip(i : int) : void`
- `+flip(i : int, j : int) : void`
- `+get(i : int) : boolean`
- `+get(i : int, j : int) : BitSet`
- `+hashCode() : int`
- `+intersects(set : BitSet) : boolean`
- `+isEmpty() : boolean`
- `+length() : int`
- `+nextClearBit(i : int) : int`
- `+nextSetBit(i : int) : int`
- `+or(set : BitSet) : void`
- `+set(i : int) : void`
- `+set(i : int, value : boolean) : void`
- `+set(i : int, j : int) : void`
- `+set(i : int, j : int, value : boolean) : void`
- `+size() : int`
- `+toString() : String`
- `+xor(set : BitSet) : void`
The HashSet solution

For each bit, 7 32-bit words overhead

or

For 64 bits, ~448 32-bit words overhead
The HashMap solution

keyValue = bitvalue / 64

bits = packed 64 bits (63 – 0)

For 64 bits, ~8 32-bit words overhead
The (homegrown) Hashtable solution

- **Key**: An int, \( \text{bitvalue} / 64 \)
- **Value**: A long, packed 64 bits (63–0)
- For 64 bits, \(~4\) 32-bit words overhead
The “Bucket by Masking” Problem

- HashMap uses masking by powers of 2 to obtain a bucket number.
- If clumping occurs, it remains when the bucket table is resized.
- Replaced masking by remaindering modulo the given table of primes, for table sizes of $2^n$.
- Every table resize returned average list length to 1.1.

- Primes $\leq 2^n$ ($n = 0, 25$)
  - $1, 2, 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381, 32749, 65521, 131071, 262139, 524287, 1048573, 2097143, 4194301, 8388593, 16777213, 33554393$
The Assessment and Response

JUnit tests (with timers) were built to test every method (verified against BitSet).

The speed of operation seems good, but this effort started with concern over memory usage.

Other problems:

- Sequential access is less than efficient: hash computation is done to find the “next” 64-bits.
- Bucket lists can be very long, and this destroys performance.

Perhaps time for some real research into the theory—
The Theory

An alternative to the list is a bit vector representation of sets. Assume the universe of discourse $U$ (of which all sets are subsets) has $n$ members. Linearly order the elements of $U$. A subset $S \subseteq U$ is represented as a vector $v_S$, of $n$ bits, where the $i$th bit in $v_S$ is 1 if and only if the $i$th element of $U$ is an element of $S$. We call $v_S$ the characteristic vector of $S$.

The bit vector representation has the advantage that one can determine whether the $i$th element of $U$ is an element of a set in time independent of the size of the set. Furthermore, basic operations on sets such as union and intersection can be carried out by the bit vector operations $\lor$ and $\land$.
“Virtual Memory”

After some thought, decided to try (as an experiment) a “virtual memory” approach

This thought pattern evolved over a period of not touching the code, and the process of evolution was neither science nor engineering (otherwise, a true "hack").

In a “virtual memory” scenario, a table of virtual addresses is kept, that point to real memory if that memory is actually needed, and do not point anywhere if that memory is not actually in use.
The first “virtual memory” structure
The final “virtual memory” structure

LEVEL1  LEVEL2  LEVEL3  6

long[][]

long[][]

table

long[]

area

long[]

block
Example: code based on “virtual memory”: flip

public void flip(int i)
{
    if( (i + 1) < 1  )throw new IndexOutOfBoundsException("i=" + i);
    final int w = i >> SHIFT3;
    final int w1 = w >> SHIFT1;
    final int w2 = (w >> SHIFT2) & MASK2;

    if( i >= bitsLength ) resize(i);
    long[][] a2;
    if( (a2 = bits[w1]) == null )
        a2 = bits[w1] = new long[LENGTH2][];
    long[] a3;
    if( (a3 = a2[w2]) == null )
        a3 = a2[w2] = new long[LENGTH3];
a3[w & MASK3] ^= 1L << i;  //Flip the designated bit
    cache.hash = 0;        // Invalidate size, etc., values
}
Example: code based on “virtual memory”: clear

```java
public void clear(int i) {
    /* In the interests of speed, no check is made here on whether
     * the level3 block goes to all zero. This may be found and
     * corrected in some later operation. */
    if( (i + 1) < 1 ) throw new IndexOutOfBoundsException("i=" + i);
    if( i > bitsLength ) return;
    final int w = i >> SHIFT3;
    long[][] a2;
    if( (a2 = bits[w >> SHIFT1]) == null ) return;
    long[] a3;
    if( (a3 = a2[(w >> SHIFT2) & MASK2]) == null ) return;
    a3[w & MASK3] &= ~(1L << i); // Clear the indicated bit
    cache.hash = 0; // Invalidate size, etc.,
}
```
Example: code based on “virtual memory”: and

```java
for( int w1 = 0; w1 != length1; ++w1 ) {
    if( (b_level2 = b_level1[w1]) == null ) level1[w1] = null;
    else if( (level2 = level1[w1]) != null ) {
        boolean level2_area_is_all_null = true;  // Pre-assumption
        for( int w2 = 0; w2 != LENGTH2; ++w2 ) {
            if( (b_level3 = b_level2[w2]) == null ) level2[w2] = null;
            else if( (level3 = level2[w2]) != null ) {
                final int index_base = (w1 << SHIFT1) + (w2 << SHIFT2);
                boolean level3_block_is_all_zero = true;
                for( int w3 = 0; w3 != LENGTH3; ++w3 )
                    if( (word = (level3[w3] &= b_level3[w3])) != 0 )
                        level3_block_is_all_zero = false;
                if( level3_block_is_all_zero ) level2[w2] = null;
            }
            if( level2[w2] != null ) level2_area_is_all_null = false;
        }
        if( level2_area_is_all_null ) level1[w1] = null;
    }
}
```
Looking at the way blocks were allocated and discarded, a side project was mounted to investigate keeping of pool of blocks:

- A stack where discarded blocks were placed (to a given limit). Each discarded block had to be guaranteed empty.
- From which needed blocks were pop’ed as needed, and if the stack ran out, new blocks were constructed.

A program was constructed to time this versus simple discarding with garbage collection, and allocation upon need.
## New Allocation vs. Pooling Blocks

<table>
<thead>
<tr>
<th>Count of allocations/discards</th>
<th>Thread time(ms) (2 sig. digits)</th>
<th>Elapsed time(ms) (2 sig. digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Using new</td>
<td>“pooled”</td>
</tr>
<tr>
<td>100,000</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>110</td>
<td>60</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1000</td>
<td>600</td>
</tr>
<tr>
<td>100,000,000</td>
<td>10000</td>
<td>6000</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>100000</td>
<td>60000</td>
</tr>
</tbody>
</table>

All timings in this presentation were made on a Lenovo ThinkPad, with an Intel Core 2 DUO CPU T9600, 2.80 GHz, with 3GB RAM, under Windows XP SP3.
Example: code based on “virtual memory”: and

```c
for( int w1 = 0; w1 != length1; ++w1 ) {
    if( (b_level2 = b_level1[w1]) == null ) level1[w1] = null;
    else if( (level2 = level1[w1]) != null ) {
        boolean level2_area_is_all_null = true; // Pre-assumption
        for( int w2 = 0; w2 != LENGTH2; ++w2 ) {
            if( (b_level3 = b_level2[w2]) == null ) level2[w2] = null;
            else if( (level3 = level2[w2]) != null ) {
                final int index_base = (w1 << SHIFT1) + (w2 << SHIFT2);
                boolean level3_block_is_all_zero = true;
                for( int w3 = 0; w3 != LENGTH3; ++w3 )
                    if( (word = (level3[w3] &= b_level3[w3])) != 0 )
                        level3_block_is_all_zero = false;
                if( level3_block_is_all_zero ) level2[w2] = null;
            }
            if( level2[w2] != null ) level2_area_is_all_null = false;
        }
        if( level2_area_is_all_null ) level1[w1] = null;
    }
}
```

Copyright © 2010 Paladin Software Incorporated, Inc.
Hand Waving

- The next part is difficult to describe.
- Problem: lots of very similar, repeated code.
- Try a management method for the “whole set” problems, with flags and switches for the various kinds of operations.
- Similarly, for the “scanning set” operations.
- Merge these, by treating “whole set” as scanning from 0 to bitsLength.
- Bugs galore in trying to get all the “special” cases to work (everything is fragile).
- Start to suspect that need to walk code over the structure (Visitor Pattern?).
Design Patterns

- “Visitor” turned out to be the wrong concept.
- **Strategy** was the needed pattern.
  - One management routine to walk the structure, and maintain it.
  - At the critical points in the walk, invoke the method of a Strategy to perform the appropriate actions.
AbstractStrategy (UML)

#F_OP_F_EQ_F : const int
#F_OP_X_EQ_F : const int
#X_OP_F_EQ_F : const int
#X_OP_F_EQ_X : const int

#properties() : int;
#start(b : SparseBitSet) : void
#word(base : int , u3 : int, a3 : long[],
   b3 : long[], mask : long) : boolean
#block(base : int, u3 : int, v3 : int, a3 : long[],
   b3 : long[]) : boolean
#void finish(a2Count : int, a3Count: int )
#isZeroBlock(a3 : long[]) : boolean

The Strategies
- AndStrategy
- AndNotStrategy
- ClearStrategy
- CopyStrategy
- EqualsStrategy
- FlipStrategy
- IntersectsStrategy
- OrStrategy
- SetStrategy
- UpdateStrategy
- XorStrategy
A Strategy

protected class AndStrategy extends AbstractStrategy {
    protected int properties() {
        return F_OP_F_EQ_F + F_OP_X_EQ_F + X_OP_F_EQ_F;
    }
    protected void start(SparseBitSet b) {
        if (b == null) throw new NullPointerException();
        cache.hash = 0;
    }
    protected boolean word(int base, int u3, long[] a3, long[] b3, long mask) {
        return (a3[u3] &= b3[u3] | ~mask) == 0L;
    }
    protected boolean block(int base, int u3, int v3, long[] a3, long[] b3) {
        boolean isZero = true;  // Presumption
        for (int w3 = u3; w3 != v3; ++w3) isZero &= ((a3[w3] & b3[w3]) == 0L);
        return isZero;
    }
}
And now the code!

```java
public void and(SparseBitSet b)
{
    nullify(Math.min(bits.length, b.bits.length));  // Optimisation
    setScanner(0, Math.min(bitsLength, b.bitsLength), b, andStrategy);
}

public void andNot(int i, int j, SparseBitSet b)
throws IndexOutOfBoundsException
{
    setScanner(i, j, b, andNotStrategy);
}

public void flip(int i, int j) throws IndexOutOfBoundsException
{
    setScanner(i, j, null, flipStrategy);
}
```
**LEVEL2 and LEVEL3 sizes**

<table>
<thead>
<tr>
<th>Level 2</th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>12.5</td>
<td>7.6</td>
<td>6.1</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>64</td>
<td>8.8</td>
<td>8.0</td>
<td>5.5</td>
<td>4.9</td>
<td>4.9</td>
</tr>
<tr>
<td>32</td>
<td>9.9</td>
<td>7.0</td>
<td>6.8</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>16</td>
<td>11.5</td>
<td>8.5</td>
<td>7.6</td>
<td>7.2</td>
<td>7.2</td>
</tr>
<tr>
<td>8</td>
<td>15.1</td>
<td>12.5</td>
<td>11.4</td>
<td>11.4</td>
<td>10.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level 3</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>
**BitSet** (UML)

- BitSet()
- BitSet(int : nbits)
- and(i : int, j : int, b : SparseBitSet) : void
- andNot(i : int, j : int, b : SparseBitSet) : void
- or(i : int, j : int, b : SparseBitSet) : void
- statistics() : String
- statistics(String[] values) : String
- toStringCompaction(boolean change) : void
- toStringCompaction(int count) : void
- xor(i : int, j : int, b : SparseBitSet) : void

- Made possible the easy addition of bounded operations (almost impossible to compose using the given operations).
- Added methods to report the internal statistics of the class (to help with memory management).
- Added an option to shorten the toString output (for “(1, 2, 3, 4, 8)” give “(1..4, 8)”.

Copyright © 2010 Paladin Software Incorporated, Inc.
So How Did It Do?

<table>
<thead>
<tr>
<th>Set/flip/clear 1</th>
<th>BitSet</th>
<th>Sparse (hash)</th>
<th>%</th>
<th>Sparse (VM)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.42</td>
<td>5.16</td>
<td>80</td>
<td>4.53</td>
<td>71</td>
</tr>
<tr>
<td>Set/flip/clear 2</td>
<td>2.13</td>
<td>4.00</td>
<td>188</td>
<td>2.55</td>
<td>120</td>
</tr>
<tr>
<td>Get (bulk)</td>
<td>26.00</td>
<td>8.58</td>
<td>33</td>
<td>1.48</td>
<td>6</td>
</tr>
<tr>
<td>Random set/get/clear individual bits</td>
<td>6.50</td>
<td>6.14</td>
<td>94</td>
<td>5.67</td>
<td>87</td>
</tr>
<tr>
<td>Performance comparison</td>
<td>24.93</td>
<td>1.97</td>
<td>8</td>
<td>6.76</td>
<td>27</td>
</tr>
</tbody>
</table>

Set/flip/clear 1: Sets bits 0 to 100,000,000 individually, and flips them, then sets them again, and clears them.

Set/flip/clear 2: Bulk sets bits 0 to 1,000,000, and bits 4,000,000 to 10,000,000, bulk flips them, the does it again, and clears them, repeated 300 times.

Get(bulk): Gets a new set (with bits 0 to 100,000 and bits 9,900,000 to 10,000,000 set) from an existing set 20,000 times.

Random etc.: Sets random bits in the range 0 to 10,000, 10,000 times, does a get on each bit, and then clears it, the whole sequence done 8,000 times.

Performance: A mixture of operations, including construction, setting, getting, clearing, doing various and, or, and xor operations, checking cardinality and hashCode, cloning, and finding intersections, done 100,000 times.
And this started with memory concerns

For $2^{26}$ 64-bit words the overhead is approximately:

- $2^{15}$ 32-bit pointers to areas
- $2^{15}$ areas with 32 32-bit pointers plus 1 word object overhead
- $2^{20}$ words of overhead across all the blocks.

\[
\begin{align*}
32768 \\
1081344 \\
1048576 \\
2162688
\end{align*}
\]

which gives $\sim0.03$ 32-bit words overhead per 64 bits.
The Experience

- There is no way that the final structure and algorithmic behavior of this component could have been predicted, designed, or otherwise anticipated at the beginning of the project.

- The Computer Science contributed as a source of techniques and their properties: hashing, hash tables, linked lists, bit vectors, object pooling, and design patterns.

- The engineering practices of constant testing, prototyping, performance and resource measurement, benefit trade-offs, tooling, and version tracking both indicated and constrained choices.

- But these disciplines do not capture the entirety of the experience!
The Bottom Line

- Developing effective and correct software (applications, components, libraries, etc.) requires understanding the possible—this is the Computer Science.

- Developing efficient and useful software is a trade-off between many issues (more than just space and time, resources, and cost), that constitute the practice of Software Engineering.

- But there remains that activity that lies between inspiration and perspiration, which is not Computer Science, and which is not Software Engineering, for which no name but “hacking” currently exists.

- The challenge, here, and elsewhere, is how to teach this skill (?), and how to bring new programmers to the place where they experience the reward.
Please, Ask questions!

Thank you!
Is it Computer Science, Software Engineering, or Hacking?

Dr. Bruce K. Haddon

Bruce.Haddon@colorado.edu