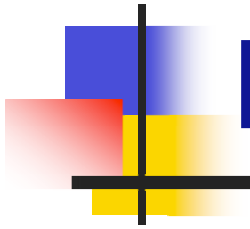


Performance Engineering From Scratch

Matthew Wakeling

FlyMine Group, Department of Genetics

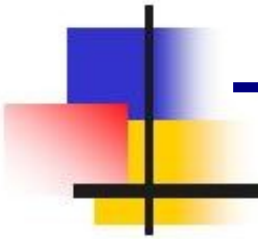
University of Cambridge

matthew@flymine.org

Colorado Software Summit

Purpose of this talk

- Why teach this stuff?
  - These are the fundamental academic case studies of well-designed algorithms.
  - Every programmer should know them.
  - They will change how you write programs.
  - They will provide a great source of ideas for improvements to your own projects.
  - Save re-inventing the wheel.
  - I aim to make you think “Wow, that's clever” at least once.



Topics

- Performance theory
  - Big-O notation
  - Performance hierarchy
- Algorithms
  - Sorting algorithms
  - Lookup algorithms
- Multiprocessing issues
- Databases
- Funky stuff

Aims

- Create awareness of general performance theory with well-understood (solved) examples.
- Examine real-world useful techniques for performance improvements.
- While away any remaining time with thought-provoking performance problems and solutions

Big-O Notation

- An indication of how much resource an algorithm will use.
  - Can be memory, CPU time, or others.
- Indicates how the resource usage scales with the size of the input data.
- Is the highest order element only.
- Applies to the average case unless otherwise specified.

Big-O Notation

- An example:
  - Say an algorithm takes a list of "n" elements and does something with the list.
  - Say it takes $28 + 14n + 7n^2$ milliseconds to run.
  - The highest order element is n^2.
  - The lower order elements are ignored.
  - The constants are ignored.
  - Therefore the algorithm is **$O(n^2)$** in time.

Big-O Notation

- Big-O Notation tells you how bad the algorithm gets when "`n`" becomes really big.
- It doesn't tell you how long the algorithm takes to run – just how it scales.
- It reveals something about the structure of the algorithm.

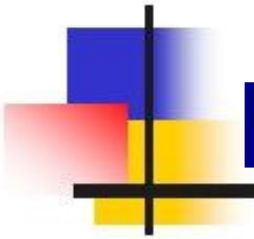
Big-O Notation

- Big-O Notation usually describes the *average* case, but can describe the *worst* case.
  - For example, a hash table has an access time of **O(1)**, but in the worst case can have an access time of **O(n)**.
  - If your application is real-time, you need an algorithm that has a "good" worst case.
  - Otherwise, you can assume worst case is rare, and have a better algorithm in the average case.

Big-O Notation

- An example of **O(n)**:

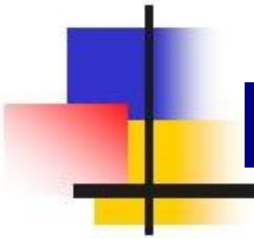
```
public static int sum(int[] array) {
        int sum = 0;
        for (int i : array) {
            sum += i;
        }
        return sum;
}
```



Big-O Notation

- An example of **O(n²)**:

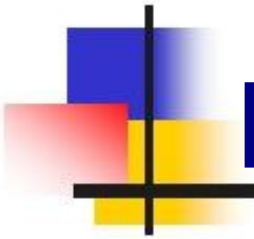
```
public static boolean duplicates(int[] array) {
        boolean duplicates = false;
        out:
        for (int i : array) {
            for (int o : array) {
                if (i = o) {
                    duplicates = true;
                    break out;
                }
            }
        }
        return duplicates;
}
```



Big-O Notation

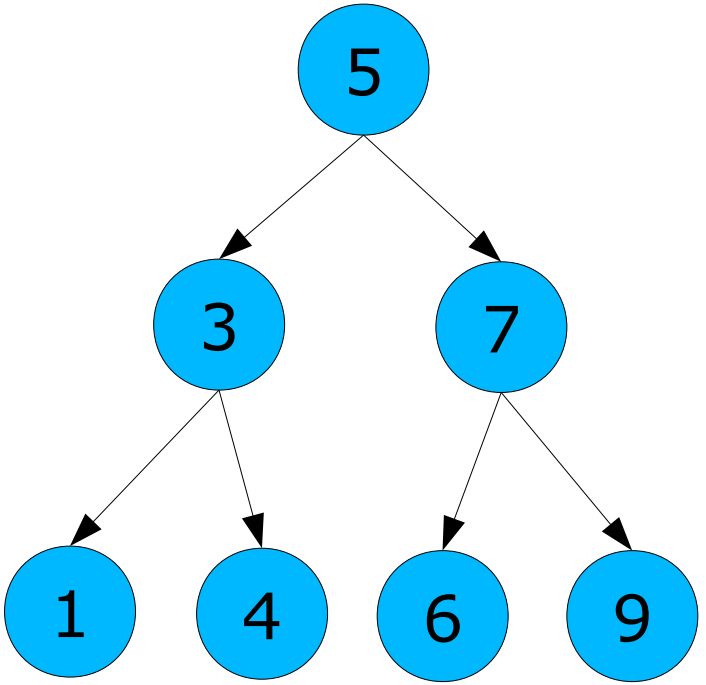
- An example of **$O(n^{0.5})$**:

```
public static boolean isPrime(int candidate) {
        boolean prime = true;
        int sqrt = Math.sqrt(candidate);
        out:
        for (int i = 2; i <= sqrt; i++) {
            if (candidate % i == 0) {
                prime = false;
                break out;
            }
        }
        return prime;
}
```



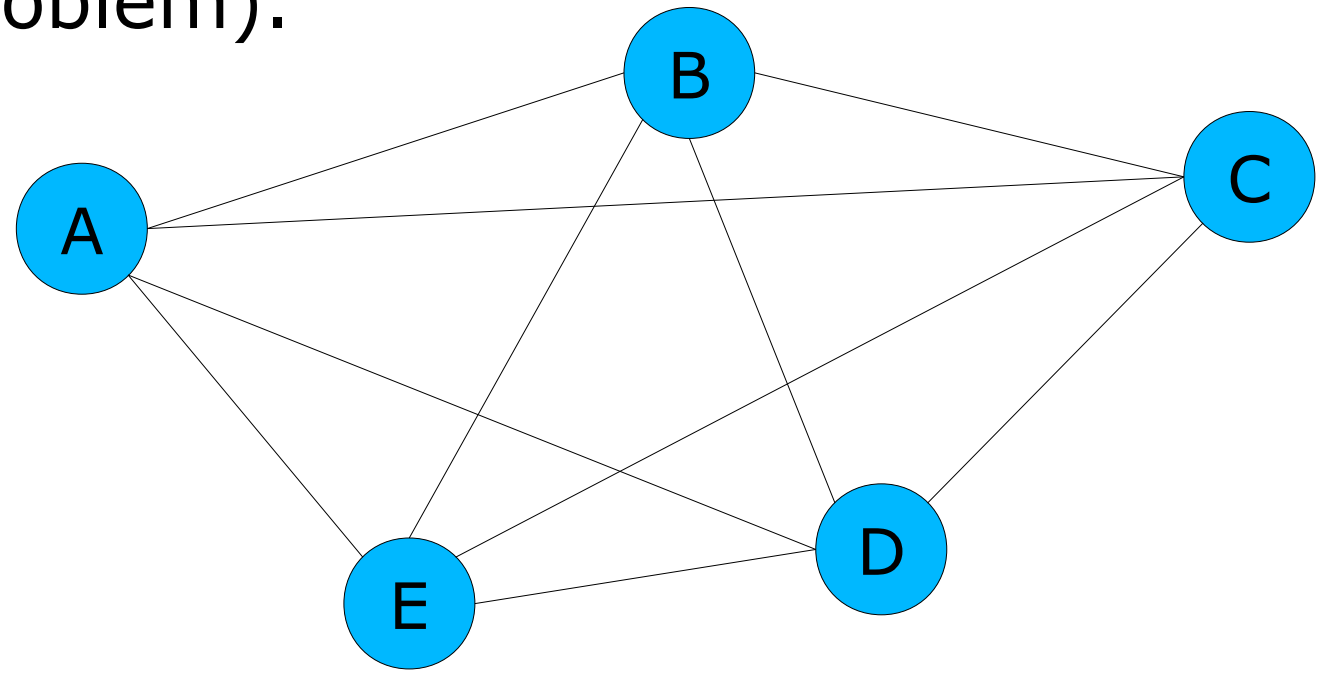
Big-O Notation

- An example of **O(log(n))**:
  - Find an entry in a balanced tree.



Big-O Notation

- An example of **O(e^n)**:
  - Find the shortest route that visits all locations (the Travelling Salesman problem).



Big-O Notation

- Beware of **O(eⁿ)** problems!
  - The Travelling Salesman problem is described as NP-complete.
  - The best algorithm found so far is **O(eⁿ)**, but it has not been proven that a better algorithm is not available.
  - However, it has been proven that if you can solve any NP-complete problem in less than **O(eⁿ)**, then you can solve all the others and all NP problems as well.
  - Moral: Not all problems can be solved!

Big-O Notation

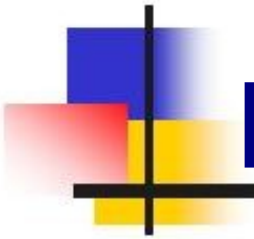
- So, you have an NP-complete problem. What do you do?
  - Fudge it! It's only NP-complete to find the guaranteed shortest route. You can find a short-ish route very easily.
  - See if there is a parameter that can be fixed to make the problem simpler.
  - Find a way to make the algorithm work for "most" input data, but fail with other data.

Big-O Notation

- Public key encryption is designed to be provably hard to crack.
  - The key has two parts – public and private.
  - You can generate the public part from the private part in a cheap operation.
  - However, it is an **$O(e^n)$** operation to generate the private key from the public key, where n is the number of bits in the key.
  - If a quicker algorithm is found, then we can solve the Travelling Salesman too!

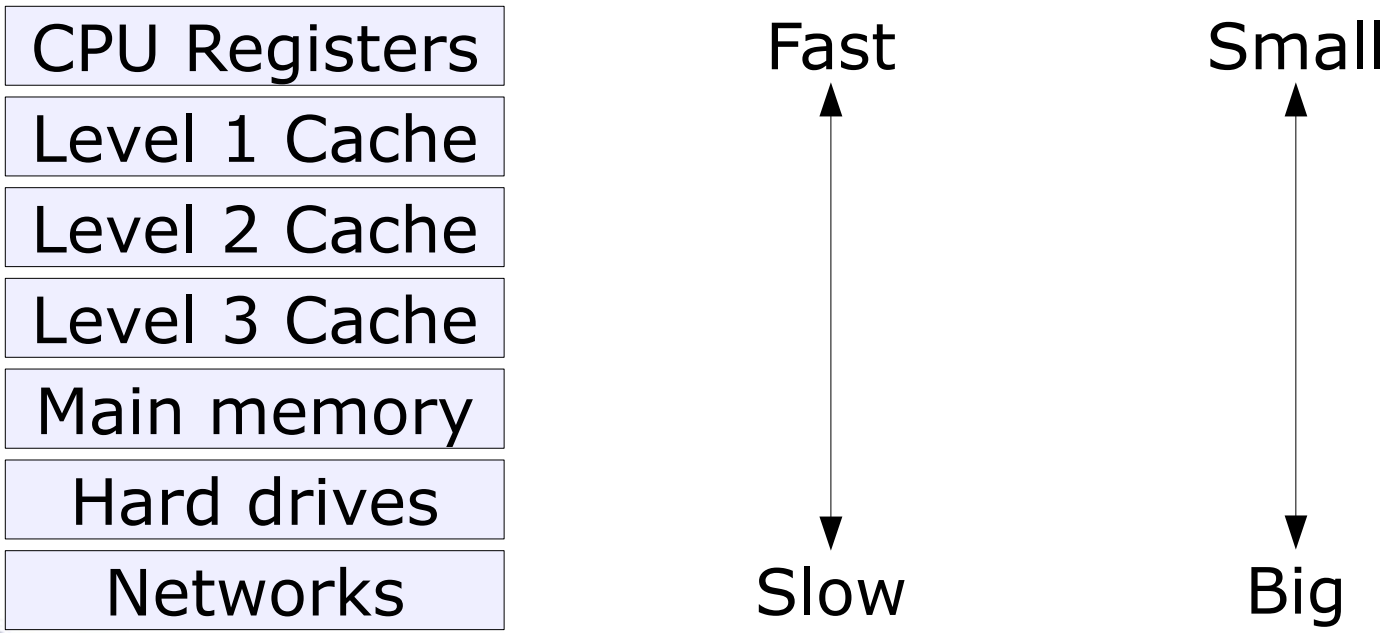
CPU Architecture

- Modern CPUs are incredible.
  - Execute multiple instructions per clock.
  - But only if they have all the data ready!
- Several things can prevent this:
  - The data is not in the cache.
  - An unexpected branch was made, and the instructions aren't in the cache.
  - An instruction uses the result of the previous instruction, which is still being executed.



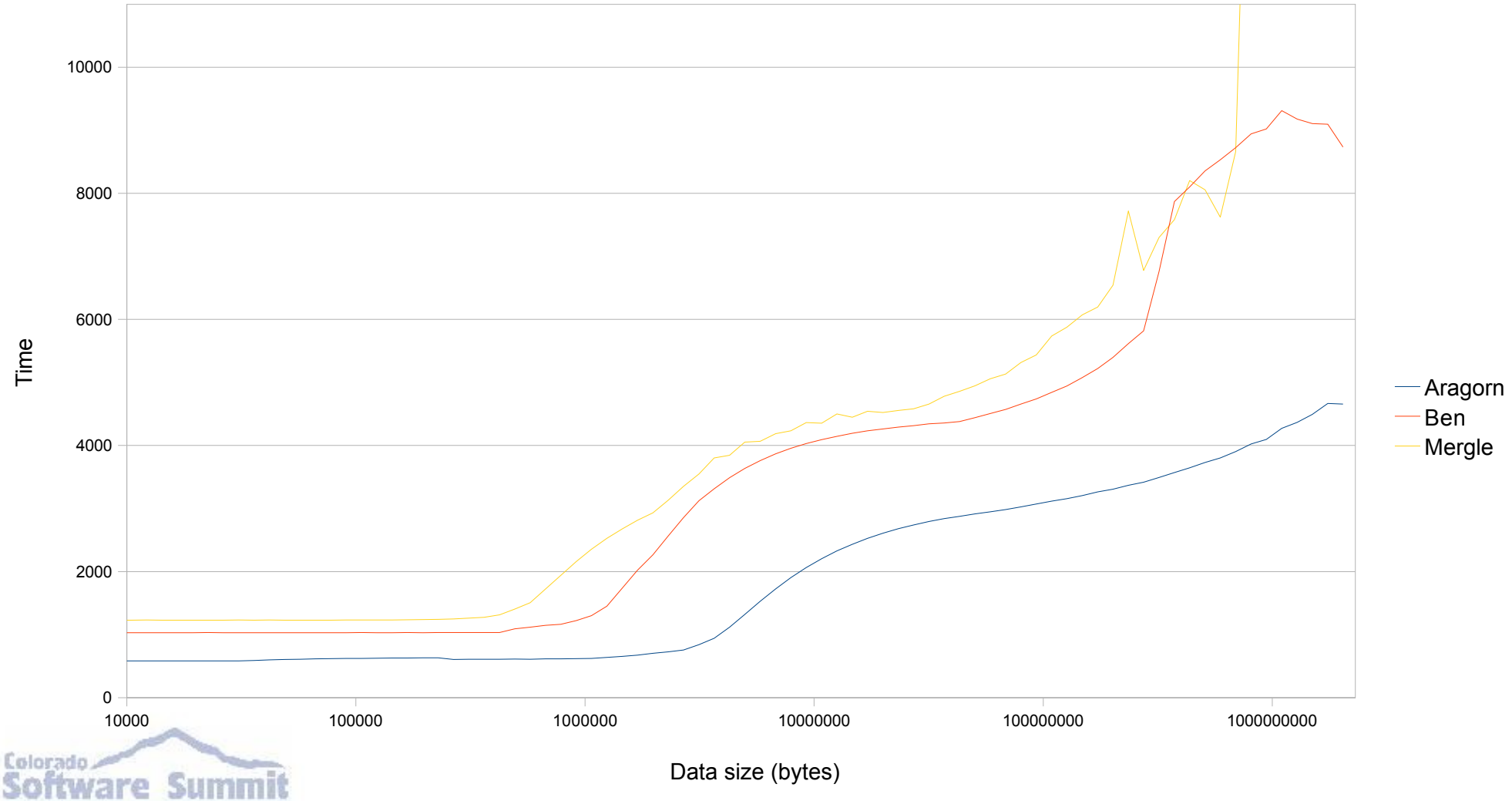
Hierarchy of Data Storage

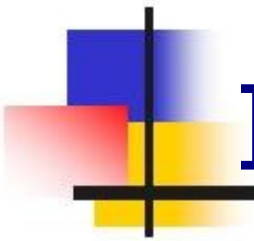
- Computers have levels of storage:
  - The larger your active data set, the slower access will be, as data has to be fetched from slower storage systems.



Hierarchy of Data Storage

- Comparative access time:





Interesting Hardware

- Some problems can be tackled by alternative hardware.
  - Usually restricted to problems that can be parallelised.
    - Physics simulations, graphics rendering, and encryption cracking.
    - Off-the-shelf graphics cards can perform some operations hundreds of times faster than general-purpose CPUs.
    - The Folding@home project receives most of its contributions from PlayStation 3 "Cell" processors. (http://folding.stanford.edu/)

Latency Versus Throughput

- Performance can be limited by latency or throughput.
  - Latency-bound means the bottleneck is waiting for round-trip times.
  - Throughput-bound means that most of the resources are being utilised to solve the problem – overheads are small.
  - Utilisation can be improved by:
    - Making larger requests to reduce overhead.
    - Performing lots of requests in parallel, so the resources can stay busy while waiting for round-trips.

Algorithms

- The ultimate reference books:

  Data Structures and Algorithms

  Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft

  The Art of Computer Programming

  Donald Knuth

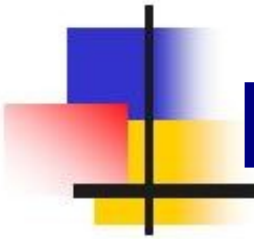
- Topics:
  - Prime number finding algorithms
  - Array sorting algorithms
  - Lookup algorithms
  - String concatenation

Prime Number Algorithms

- Task: Find all prime numbers up to "n"

```
public static List<Integer> primes(int n) {
     List<Integer> primes = new ArrayList<Integer>();
     for (int i = 2; i < n; i++) {
          boolean prime = true;
          for (int o = 2; o <= i; o++) {
               if (i % o == 0) {
                    prime = false;
               }
          }
          if (prime) {
               primes.add(i);
          }
     }
     return primes;
}
```

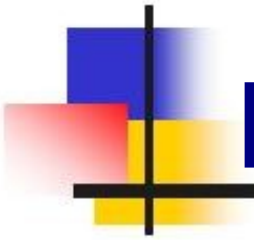
O(n²)



Prime Number Algorithms

```
public static List<Integer> primes(int n) {
     List<Integer> primes = new ArrayList<Integer>();
     for (int i = 2; i < n; i++) {
          out:
          boolean prime = true;
          for (int o = 2; o <= i; o++) {
               if (i % o == 0) {
                    prime = false;
                    break out:
               }
          }
          if (prime) {
               primes.add(i);
          }
     }
     return primes;
}
```

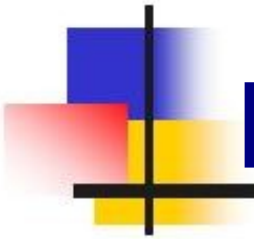
O(n²)



Prime Number Algorithms

```
public static List<Integer> primes(int n) {
     List<Integer> primes = new ArrayList<Integer>();
     for (int i = 2; i < n; i++) {
          out:
          boolean prime = true;
          int sqrt = Math.sqrt(i);
          for (int o = 2; o <= sqrt; o++) {
               if (i % o == 0) {
                    prime = false;
                    break out:
               }
          }
          if (prime) {
               primes.add(i);
          }
     }
     return primes;
}
```

$O(n^{1.5})$



Prime Number Algorithms

```
public static List<Integer> primes(int n) {
     List<Integer> primes = new ArrayList<Integer>();
     for (int i = 2; i < n; i++) {
          out:
          boolean prime = true;
          int sqrt = Math.sqrt(i);
          for (int o : primes) {
               if (i % o == 0) {
                    prime = false;
                    break out:;
               }
               if (o > sqrt) break out:;
          }
          if (prime) {
               primes.add(i);
          }
     }
     return primes;
}
```

$O(n^{1.5})$

Prime Number Algorithms

```
public static List<Integer> primes(int n) {
     boolean sieve[] = new boolean[n];
     for (int i = 2; i < n; i++) {
          sieve[i] = true;
     }
     List<Integer> primes = new ArrayList<Integer>();
     for (int i = 2; i < n; i++) {
          if (sieve[i]) {
               primes.add(i);
               for (int o = i * i; o < n; o += i) {
                    sieve[o] = false;
               }
          }
     }
     return primes;
}
```

The Sieve of Eratosthenes

O(nlog(n)log(log(n)))

276 BC - 194 BC

Array Sorting Algorithms

- There are more array sorting algorithms than you can shake a stick at.
- Not all are *stable* sorts.
  - Two equivalent elements will stay in the same order relative to each other.
- java.util.Arrays.sort() is very good.
  - However, it is educational to look at the different alternatives.

Sorting – Bubble Sort

- Very simple algorithm:

```
private static void bubbleSort(int a[]) {
     boolean swapped;
     int end = a.length - 1;
     do {
          swapped = false;
          for (int i = 0; i < end; i++) {
               if (a[i] > a[i + 1]) {
                    int swap = a[i];
                    a[i] = a[i + 1];
                    a[i + 1] = swap;
                    swapped = true;
               }
          }
          end--;
     } while (swapped);
}
```

Sorting – Bubble Sort

Sorting – Bubble Sort

- "The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems." -- Donald Knuth
- **$O(n^2)$** - Requires n scans of the array of size n.
- High values move right quickly, but low values move left slowly.
  - Hares and Tortoises.

Sorting – Cocktail Sort

- Attempt to solve the "Hares and Tortoises" problem of Bubble Sort.
- Scan in both directions, instead of just left to right.
- Still **O(n²)**.

Sorting – Selection Sort

- Lowest values are selected, and swapped into place.

```
private static void selectionSort(int a[]) {
    for (int i = 0; i < a.length - 1; i++) {
        int minPos = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[minPos]) {
                minPos = j;
            }
        }
        int swap = a[minPos];
        a[minPos] = a[i];
        a[i] = swap;
    }
}
```

Sorting – Selection Sort

- Usually faster than bubble sort.
- Performs **O(n²)** comparisons, but only **O(n)** swaps.

Sorting – Insertion Sort

- Values are inserted leftwards into the sorted values

```
private static void insertionSort(int a[]) {
    for (int i = 1; i < a.length; i++) {
        int swap = a[i];
        int j = i - 1;
        while (j >= 0 && a[j] > swap) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = swap;
    }
}
```

Sorting – Insertion Sort

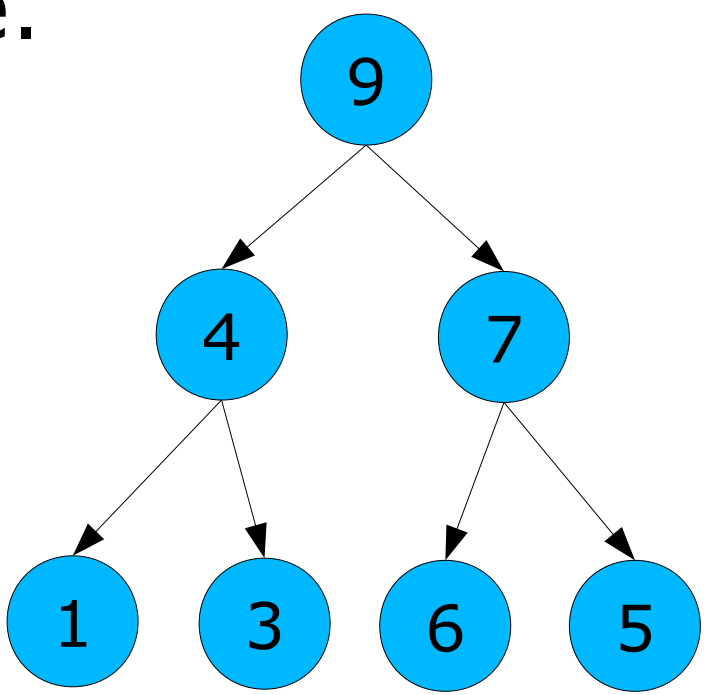
- Usually the fastest sort for 70 items or fewer.
- Still **O(n²)**.

Sorting – Heap Sort

- Heap Sort is a variation of Selection Sort, but achieves **O(nlog(n))** complexity by using a *heap*.
  - A heap is a balanced tree structure where the highest element is always at the top.
  - The sort is performed in two stages:
    - Firstly, turn the array into a heap. **O(nlog(n))**
    - Then repeatedly (**O(n)** times):
      - √ select the element at the top of the tree. **O(1)**
      - √ swap it into position. **O(1)**
      - √ fix up the heap. **O(log(n))**

Sorting – Heap Sort

- A heap can be represented in an array.
- It is always a balanced tree.
- Each node is greater than the two nodes below it in the tree.
  - The "Heap Invariant".
- Then, swap the top node with the last node.
  - The last node is then part of the "sorted" portion of the array.



Sorting – Heap Sort

Sorting – Shell Sort

- Variation on Insertion Sort, invented by Donald Shell in 1959, but achieves **O(nlog(n))** complexity by sorting with different strides.
  - Insertion sort only moves an element by one place each time.
- Rearrange the array into "x" columns, and sort each column with insertion-sort, then reduce x and repeat.
  - The best known sequence of x is 1750, 701, 301, 132, 57, 23, 10, 4, 1.

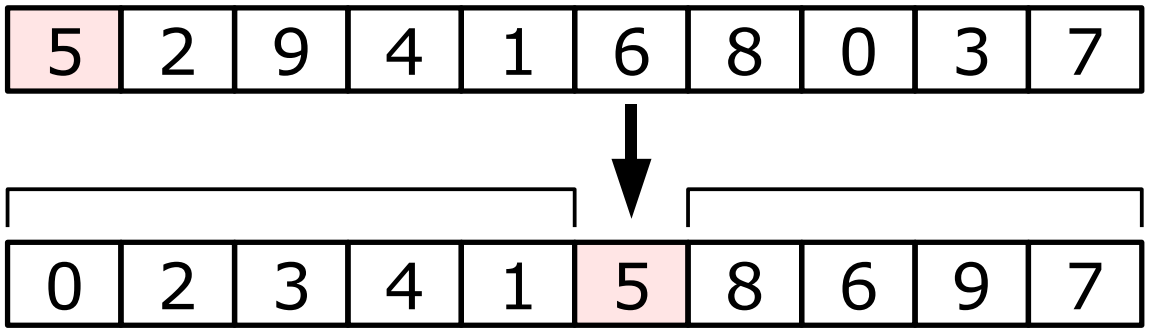
Sorting – Shell Sort

Sorting – Quicksort

- Developed by C.A.R. Hoare in 1960.
- The most commonly used sorting algorithm nowadays.
- Is **O(nlog(n))**, but has a worst-case scenario of **O(n²)**.
  - It is possible to make the worst-case scenario "unlikely", but for real-time or security applications, a different algorithm should be used.
- Not a stable sort algorithm.

Sorting – Quicksort

- To sort an array:
  - Pick one of the values, to use as a "pivot".
  - Rearrange the array so that all values lower than the pivot are to its left, and all values higher are to its right.



  - Then sort each side by recursing.

Sorting – Quicksort

Sorting – Quicksort

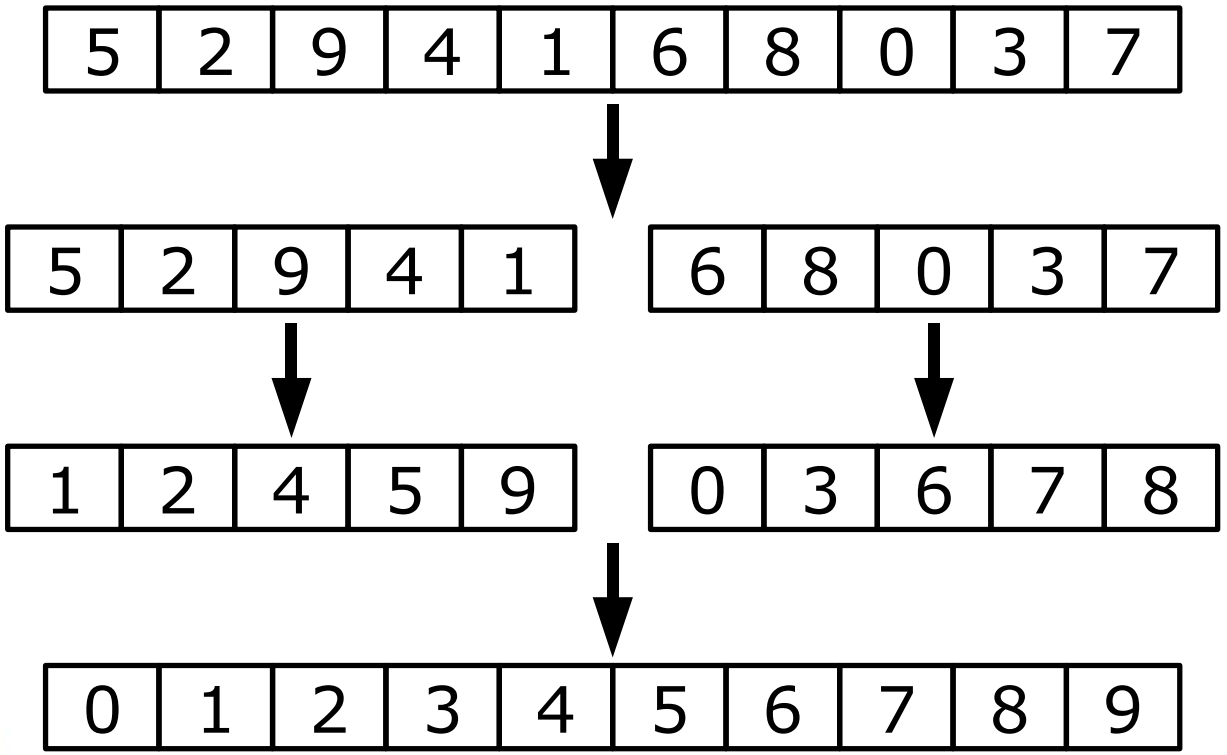
- There are two main problems:
  - Choosing the pivot. The ideal pivot is one that will divide the values into two similarly-sized groups.
    - What if the array is already sorted?
  - What to do with identical values. A variation creates three groups, placing all values equivalent to the pivot in the middle.
- An optimisation is to sort small arrays with insertion sort.

Sorting – Merge Sort

- Invented by John von Neumann in 1945.
- **O(nlog(n))**, and has no worst-case scenario.
- It is a stable sort algorithm.
- Can be parallelised.
- Used for sorting data larger than memory.
- Slightly slower than Quicksort.
- Requires **O(n)** extra storage.

Sorting – Merge Sort

- To sort an array:
  - Split the array in two, and sort both halves.
  - Merge the two halves together again.



Sorting – Merge Sort

Sorting – Bigger than memory

- What if the data to be sorted is bigger than will fit in memory?
  - Sort large "chunks" of data, and dump them into files on disc.
  - Merge all the files together using merge sort to produce a stream of sorted data.
  - Performance will be close to the sequential speed of your disc architecture.
  - Old machines used to use multiple tapes for the temporary storage.

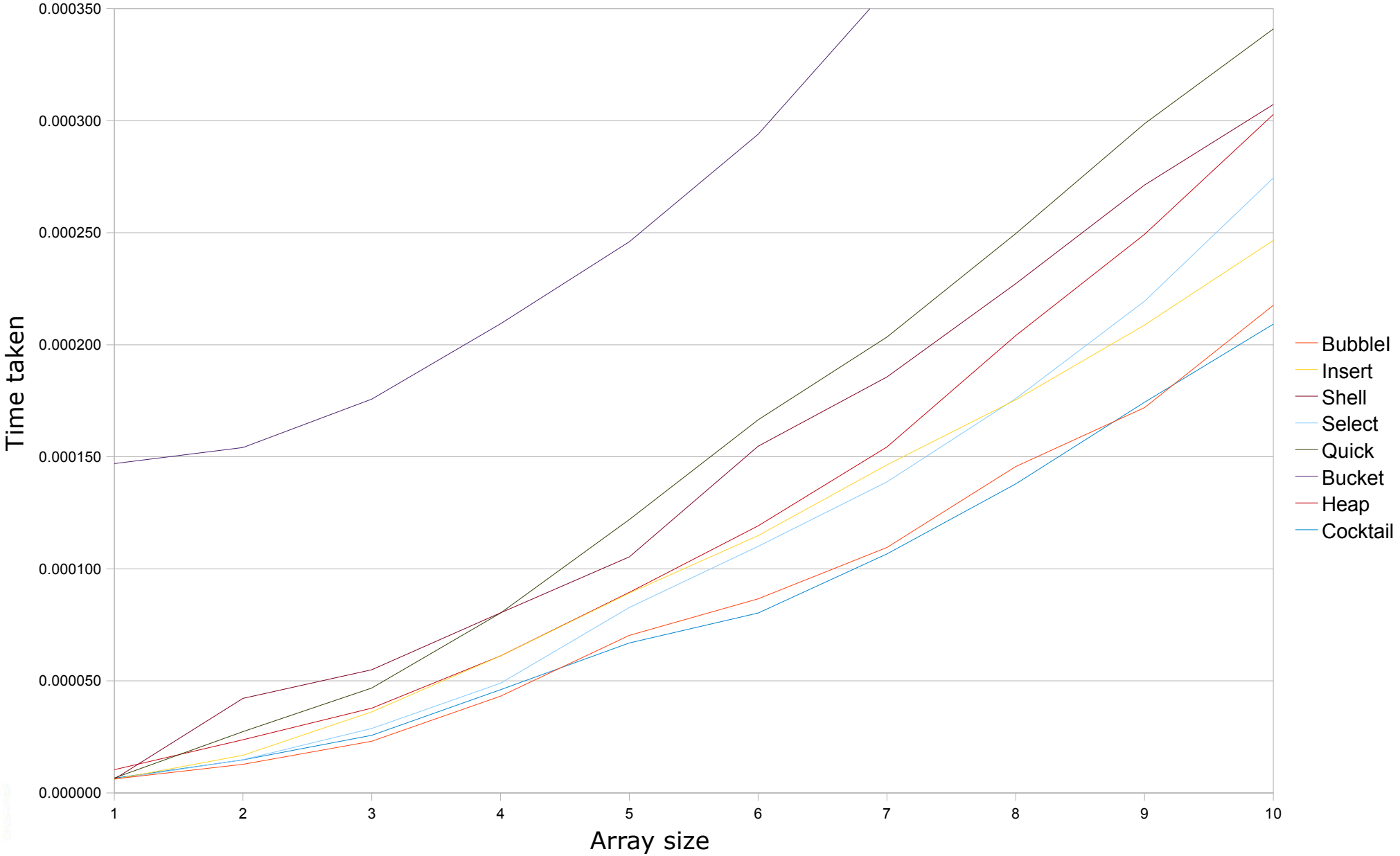
Sorting

- There are many more sorting algorithms.
  - You don't need to know them.
  - java.util.Arrays.sort() is very good – use it.
- Theory states that comparison sorts are limited to **O(nlog(n))**.
  - This limit can be broken if you have extra information about the distribution of values.

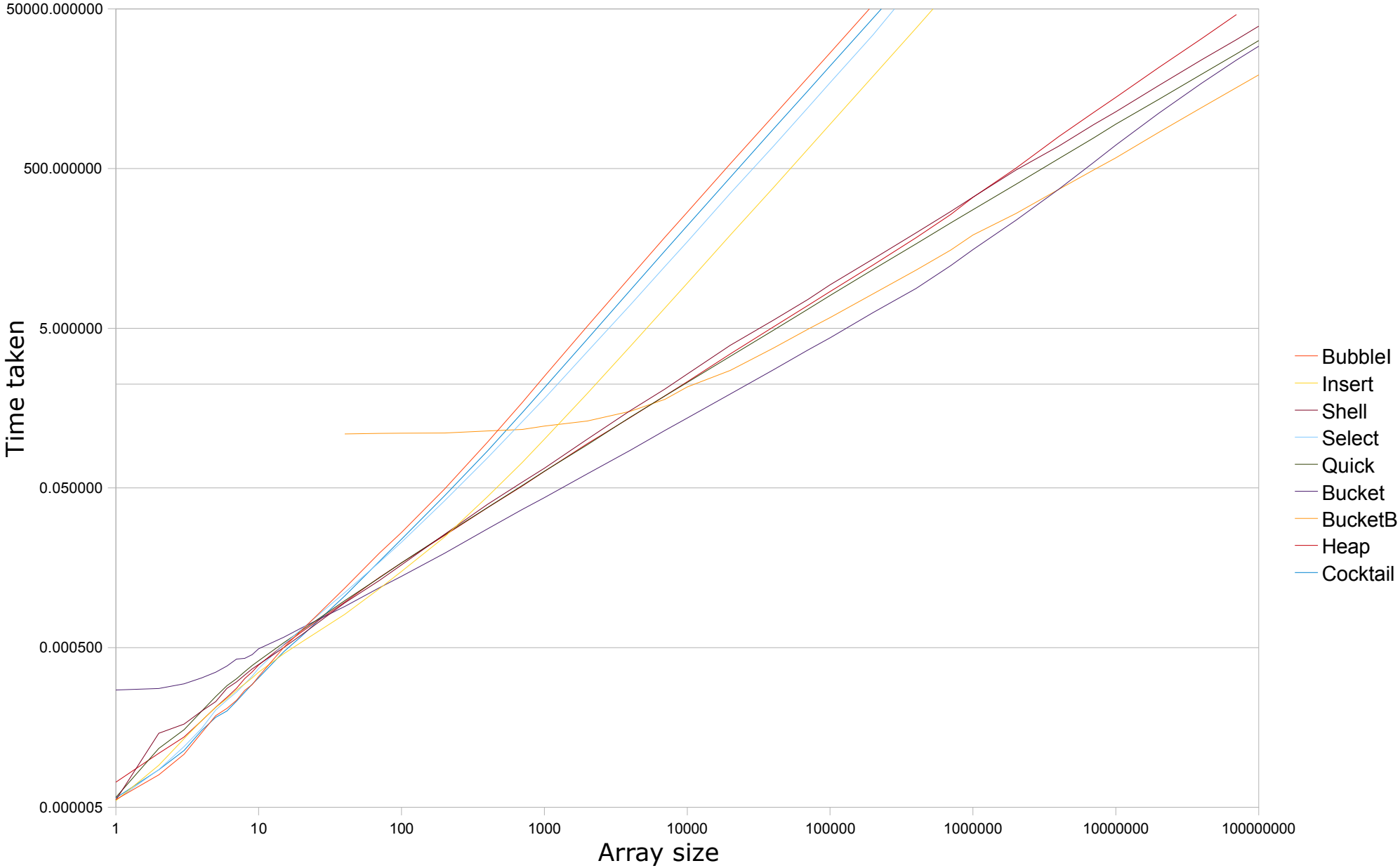
Sorting – Bucket Sort

- To sort an array of values:
  - Create a set of buckets for various ranges of values.
  - Place all the values in their buckets.
  - Sort each bucket individually.
  - Concatenate the sorted buckets in order.
- Algorithm is **O(n)**, but requires **O(n)** of extra memory.
- In reality, it doesn't play well with CPU caches.

Sorting – Comparison



Sorting - Comparison



Lookup Algorithms

- Purpose: Store data and look it up with a key. java.util.Map
- Simplest method: Array search.

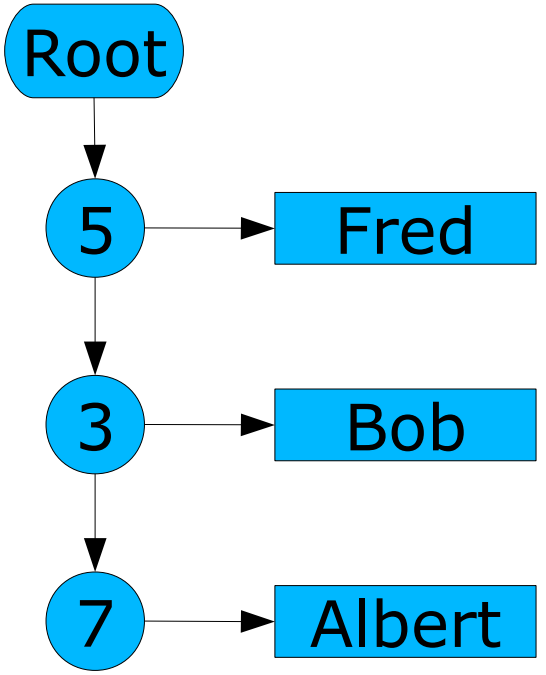
```
public static boolean isPresent(int values[], int key) {
     for (int value : values) {
          if (value == key) {
               return true;
          }
     }
     return false;
}
```

  - Lookup is **O(n)**.
  - Interesting to resize the array.

Lookup – Linked Lists

- Another linear search.
  - The basis of more complex searches.

```
public class LinkedListNode {
     private LinkedListNode next;
     private int key;
     private String value;
     ...
}
```



  - Easy to add and remove nodes.
  - Most operations are **O(n)**.
    - Including some which are **O(1)** on arrays.
    - This can cripple some sort algorithms.

Lookup – Binary Search

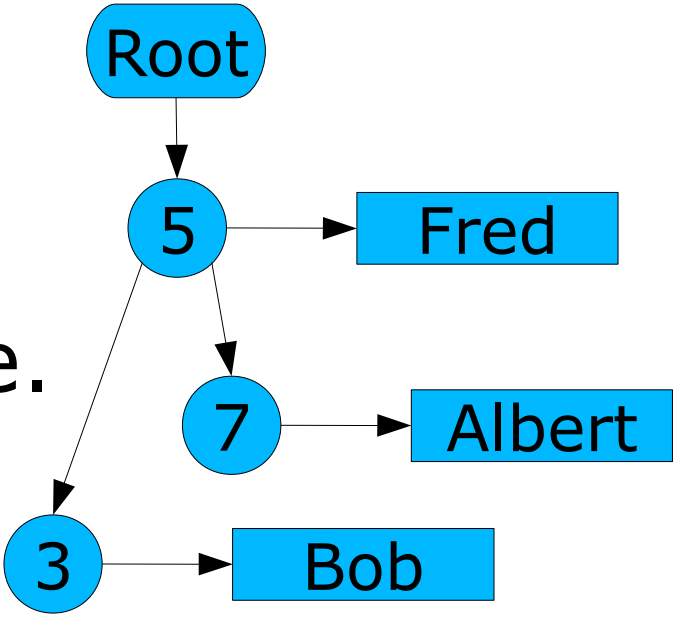
- Used if you have a sorted array.
  - A much-neglected algorithm, but it is fast.
  - http://www.tbray.org/ongoing/When/200x/2003/03/22/Binary
- Very simple algorithm:
  - You have an array.
  - You look at the middle element.
    - If it is too high, concentrate on the bottom half.
    - If it is too low, concentrate on the top half.
  - Repeat until you have found the value, or established it is not present.

Lookup – Binary Search

- Algorithm is **O(log(n))**.
  - However, since the algorithm is so simple, it often beats **O(1)** algorithms.
- You don't need to implement your own binary search algorithm.
  - Use java.util.Arrays.binarySearch()
- Updating the array is hard, as it is sorted.
- An array has zero indexing overhead, compared to other data structures.

Lookup – Tree Search

- An extension of the linked list is for the nodes to have more than one reference to other nodes, creating a tree.
  - Create some rule that distinguishes between the different references.
  - For example, in this tree, the left reference points to lower key values, and the right reference points to higher key values.
  - Searches are **O(log(n))** instead of **O(n)**.



Lookup – Tree Search

- Many types of trees exist, with their advantages and disadvantages.
  - It is important to keep your trees balanced, or the **O(log(n))** behaviour could degenerate into **O(n)**.
- java.util.TreeMap and java.util.TreeSet are good implementations of red-black trees.
  - An additional advantage of a tree is that it can be traversed to return the elements sorted.

Lookup – Hash Search

- Another extension of the linked list idea is to have multiple linked lists instead of just one.
  - Similar to the bucket sort idea.
  - Need a way to know which list to look in.
  - Run the key object through a hash algorithm, which returns an integer.
    - Object.hashCode()
  - Use this integer to index into the correct bucket.

Lookup – Hash Search

- Lookup in a Hash Table is **O(1).**
  - Given sufficient buckets.
  - Assuming the hash function spreads the keys evenly among the buckets.
- java.util.HashMap and java.util.HashSet are good implementations.
- However, the order of the elements is not preserved.
  - Could combine a linked list with a hash table, as in java.util.LinkedHashMap/Set.

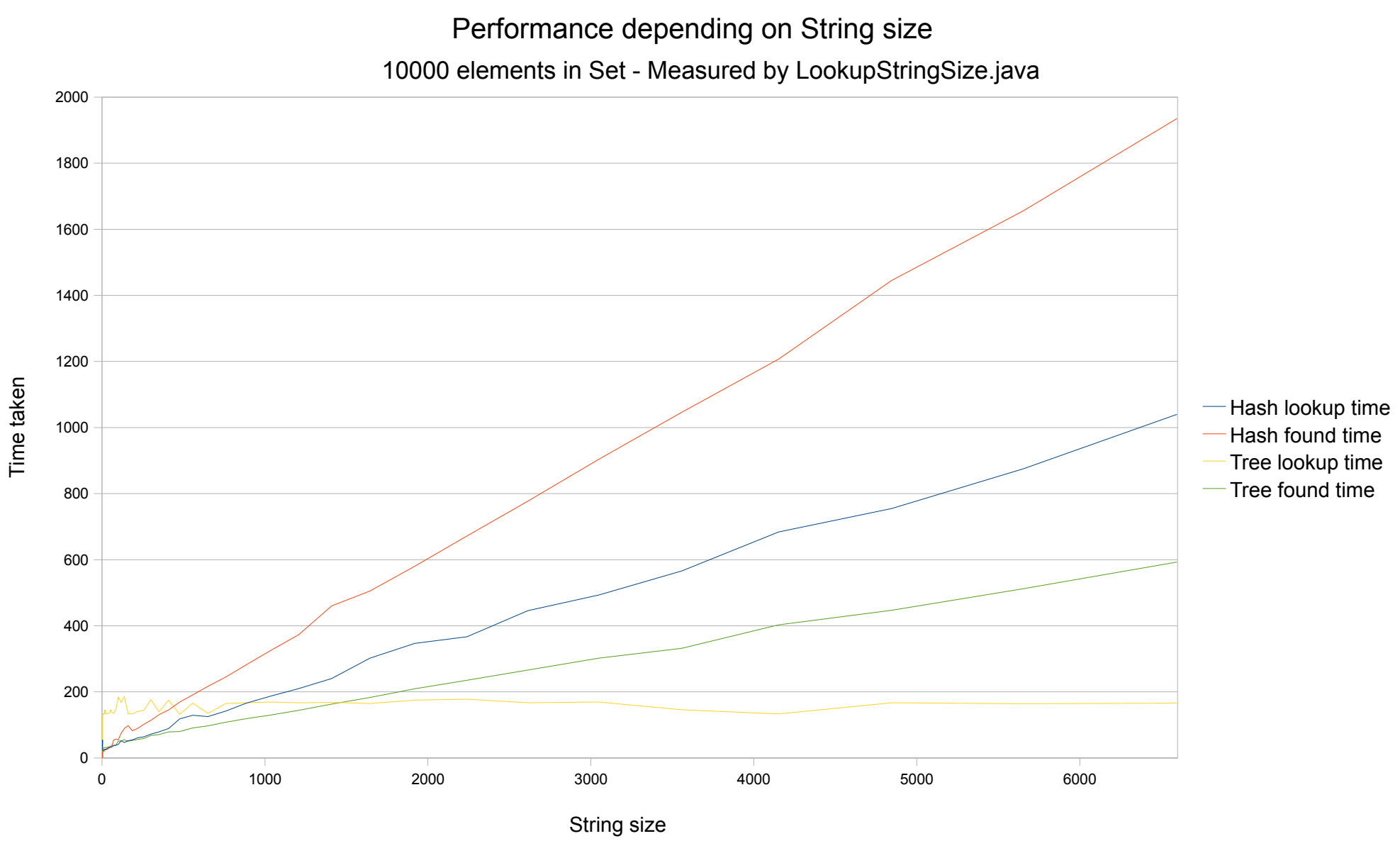
Lookup - Bitmap

- If all you want is to know if an integer is present, a very space-efficient method is a bitmap.
  - Note that Java boolean arrays do not pack bits efficiently.
  - Use integer array or java.util.BitSet instead.
- Space-efficiency is good.
  - Improves cache coherency, as you can fit more data into the cache.

A Gotcha

- So far, performance has been measured by the number of comparisons.
  - What if a comparison takes a long time?
- For example, String.equals() and String.hashCode() need to check every character of the String.
  - **O(n)** where n is the length of the String.
  - However, String.hashCode is cached.
- String.compareTo() will terminate early if the strings are not equal.

A Gotcha



String Concatenation

- The classic performance mistake is the following:

```
// Convert List of numbers into a String
String text = "";
for (int i : numbers) {
    text = text + i + " ";
}
```

O(n²)

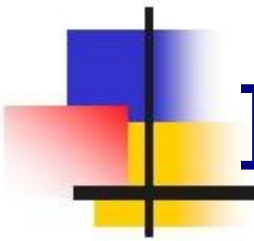
  - Instead:

```
StringBuffer textBuffer = new StringBuffer();
for (int i : numbers) {
    textBuffer.append(i + " ");
}
String text = textBuffer.toString();
```

O(n)

Memory Efficiency

- Some variable-size objects in Java double their size when they are full.
  - This can be non-optimal under some circumstances, for example if you append a 100MB String to a StringBuffer, and then append a few extra characters.
  - To save memory, try setting the capacity, or altering the behaviour by subclassing.
    - org.intermine.util.SensibleByteArrayOutputStream
    - org.intermine.util.StringConstructor



Inserting Into An Array

- Inserting into an array can be expensive.
  - To insert a single element can be **O(n)**, as all the elements to the right must be moved.
- If many inserts are expected in the same place (like in a text editor), split the array with a gap in the middle.
  - Insert in the gap (**O(1)**), and expand the gap (**O(n)**) when it gets too small.
  - Moving the gap is **O(n)** where n is the distance to move.

Multi-threading – Recap

- In Java, you can create multiple threads of execution using java.lang.Thread, which run in parallel.
  - Data needs to be passed between them, by allowing multiple threads to access the same data.
  - However, uncontrolled parallel access can lead to data corruption.
  - Java uses "synchronized" blocks to solve this problem.

Multi-threading – Recap

- Critical code that accesses an object should synchronise on that object.
  - This grabs an exclusive lock.
  - Only one thread can hold the lock at a time.
  - The thread can then perform alterations on the object that leave the object temporarily inconsistent, as long as it resolves it before releasing the lock.
  - When another thread acquires the lock, the object will be consistent.

Multi-threading – Recap

- Sometimes a thread needs to wait for data from another thread.
  - Acquire the lock on an object, and call the object's "wait" method.
  - This will suspend execution of the thread and temporarily release the lock.
  - Another thread can acquire the lock and alter the object to provide the data, then call the object's "notify" method.
  - The thread wakes up and re-acquires the lock, ready to inspect the object.

Multi-threading – Workings

- For performance reasons, each thread has its own copy of variables it is using.
  - When they are first accessed, variables are copied from main heap to the local area.
  - When the thread synchronises, the variables are copied back.
- Unless you synchronise, you *really* cannot rely on data passed from one thread to another.
- CPUs work in a very similar way.

Multi-threading – ThreadLocal

- java.lang.ThreadLocal provides a method of having independent values for each Thread.
  - A good way to completely sidestep concurrency issues.
  - For instance, use it to store connections to a database, bypasses connection pool overhead.
    - Assuming each thread only needs a single connection, and you don't have an insane number of threads.

Multi-threading

- Each synchronisation incurs an overhead.
  - Try to do much work in each synchronisation.
  - Don't leave them out!
- Lock contention can reduce concurrency.
  - Only one thread at a time can be in a synchronised block of code.
  - If most of processing is done in that code, then performance will suffer.
    - Especially on multi-CPU machines.

Multi-threading – java.util.concurrent

- Java 1.5 introduced the java.util.concurrent package, containing lots of useful classes for multi-threading performance.
  - Covering the whole lot is beyond the scope of this talk.
  - Some classes are drop-in replacements:
    - java.util.concurrent.ConcurrentHashMap
    - java.util.concurrent.ConcurrentSkipListMap
    - java.util.concurrent.ConcurrentSkipListSet
    - java.util.concurrent.ConcurrentLinkedQueue

Multi-threading - IO

- IO can be a major performance issue.
  - Only one system (often input, CPU, output) can be operating at a time per thread.
  - There are two ways to increase utilisation:
    - Have a thread per system keeping each other fed. Input IO threads need to pre-fetch. This method is good for single large operations. Operating systems will do this for you for disc IO.
    - Have lots of parallel operations going on, all doing IO independently. This method is good for interactive multi-user systems, like web sites. Service time may be bad, but throughput can be great.

Databases

- Beware that databases are very complex beasts.
  - To fully get full benefits, you need to understand how they work – *i.e.* be a DBA!
  - However, a general rule is to try and do as much work in the database as possible.
    - Databases are really performance-optimised, and can shift large amounts of data very quickly.
    - Getting the database to sum a column is much faster than fetching all the columns and summing them in Java.

Databases – Latency

- Latency becomes very important when accessing databases.
  - Your query has to be constructed, sent across the network, parsed, planned, executed, and sent back.
  - A query to fetch 1000 rows will outperform 1000 queries to fetch a single row each.
  - Prepared statements help a little.
  - Likewise, single INSERT statements can be slow – some databases provide alternatives.
    - Grouping statements in a transaction can help.

Databases - Algorithms

- Databases store tables of data.
- There are several algorithms used to query that data:
  - Sequential Scan
  - Index Scan
  - Bitmap Index Scan
  - Nested Loop Joins
  - Hash Joins
  - Merge Joins
  - Sorting
  - Aggregation
  - Hash Aggregation

Databases – Sequential Scan

- If the table is a file on disc, a sequential scan reads it in order and keeps the rows that match.

```
SELECT * FROM table WHERE field = 5;
```

  - Disc access is sequential, which is fast.
  - A CPU can usually filter the results as fast as the disc provides them.
  - However, if the table is multi-gigabyte, it can still take a very long time.
  - Rows are returned in a random order.

Databases – Index Scan

- An index is an additional structure with a table, that points to places in the table for certain values of a field.
  - An index may be a hash table or tree – any structure with better than **O(n)** access.
  - Only those parts of the table file that are pointed to need to be loaded.
  - Disc access is random, which can be slow.
  - However, loading a single row is quick.
  - Rows are returned in the order of the index.

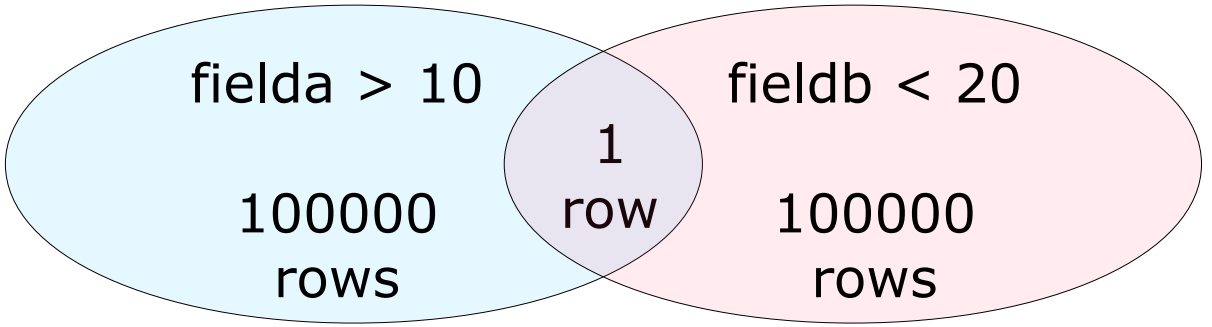
Databases – Index Scan

- Hash tables can only perform `field = 5` type lookups.
- Trees can also perform `field >= 5` type lookups, or "range" queries.
- However, the following query is hard:

```
SELECT ... WHERE fielda > 10 AND fieldb < 20
```

  - Two separate indexes for fielda and fieldb.
  - Each may restrict results by only 50%, even if the overlap is 1%.

Databases – Index Scan



- An Index Scan can only use one index.
  - Either use index on fielda and filter by fieldb, or *vice-versa*.
  - Both involve filtering 100001 rows from random disc access to get 1 row.
  - Probably faster to do a sequential scan.
- Need an R-tree index on both fields.

Databases – Bitmap Index Scan

- An attempt to get round index failings.
  - Take the results of the index (a list of places in the table), and create a bitmap.
  - Multiple bitmaps from different indexes can be combined (AND, OR).
  - Then the rows can be fetched from the table in order according to the bitmap.
  - Possibility of sequential disc access.
  - Rows are returned in random order.
  - However, the bitmap created may be big.

Databases – Nested Loop Joins

```
SELECT * FROM table1, table2 WHERE table1.fk = table2.id
```

- This query joins two tables together.
  - Returns all combinations of all rows that match.
  - The simplest way to calculate this is a nested loop:

```
for (table1 in table1_rows) {
    for (table2 in table2_rows) {
        return row if (table1.fk == table2.id);
    }
}
```

  - Very slow! **O(n²)**
  - Use an index instead for table2. **O(nlog(n))**

Databases – Hash Joins

- To improve the performance, the database could load the entire table2 into memory, and create a hash table.

```
HashTable hash = new HashTable();
for (table2 in table2_rows) {
    hash.put(table2.id, table2);
}
for (table1 in table1_rows) {
    return hash.get(table1.fk);
}
```

  - However, table2 may not fit entirely in RAM. Works well when one of the tables is small. **O(n)**

Databases – Merge Joins

- Where the two tables are sorted by the same key, a Merge Join is the fastest algorithm.
  - Read the two streams and merge, just like Merge Sort.
  - Produces sorted results in a stream. **O(n)**
  - Can be the fastest algorithm even if both tables need to be sorted first. **O(nlog(n))**
  - The database will decide which algorithm is fastest based on statistics on the data.

Reflection

- Some general-purpose code may have to access fields of classes by name.
  - Database access and XML parsers.
  - Java provides reflection to allow fields and methods of classes to be accessed.

```
Class c = Class.forName("Example");
Object o = c.getConstructor().newInstance();
c.getMethod("setName", String.class).invoke(o, "Fred");
String name = (String) c.getMethod("getName").invoke(o);
```

  - Unfortunately, reflection is rather slow.
  - You need to catch lots of exceptions.

Reflection

- Hibernate bypasses reflection for speed.
  - For each class that it is handling, Hibernate generates another class containing accessor methods, using bytecode generation.
  - Method access then goes through generated code, which can be HotSpot optimised.
  - This method is complex – you need to know how to generate bytecode.
  - Reflection is used once to generate code.
  - Provides a major speedup.

Reflection

- InterMine bypasses reflection too.
  - In InterMine, all the data classes are generated code, compiled with javac.
  - Extra methods are present in the classes to access fields by name.
  - In fact, methods exists to serialise or deserialise the objects completely.
  - Simpler to implement than bytecode generation.
  - Possibly even faster.

Reflection

- InterMine generated code.
  - All data classes implement this interface:

```
public interface FastAccessObject {
     public Integer getId();
     public void setId(Integer id);
     public Object getFieldValue(String fieldName);
     public Object getFieldProxy(String fieldName);
     public void setFieldValue(String fieldName, Object o);
     public Class getFieldType(String fieldName);
     public Class getElementType(String fieldName);
     public StringConstructor getoBJECT();
     public void setoBJECT(String notXml, ObjectStore os);
}
```

  - The OBJECT field is the serialised version.

Funky Stuff

- Topics:
  - CPU architecture
  - Simultaneous Sequential Scans (Postgres).
  - Linux 2.6.25 scheduler
  - Linux anticipatory IO scheduler
  - Sequential scans in selective buffers
  - Journalling
  - Linux network stack hash table attacks

CPU Architecture

- Back in the good old days, memory was as fast as the CPU, and memory access was simple.
  - Now, CPUs are so fast that going to memory may involve a wait of several *hundred* clock cycles.
  - CPU manufacturers are desperate to keep the CPU busy doing useful work while waiting for memory accesses.
  - Hence caches and branch prediction.

CPU Architecture

- Modern CPUs do branch prediction.
  - They "guess" which way execution will branch, based on past statistics.
  - That way, the next set of instructions can be speculatively fetched from memory before they are needed.
  - The CPU can even speculatively execute the next instructions before the branch is known, and throw away the result if it was wrong.

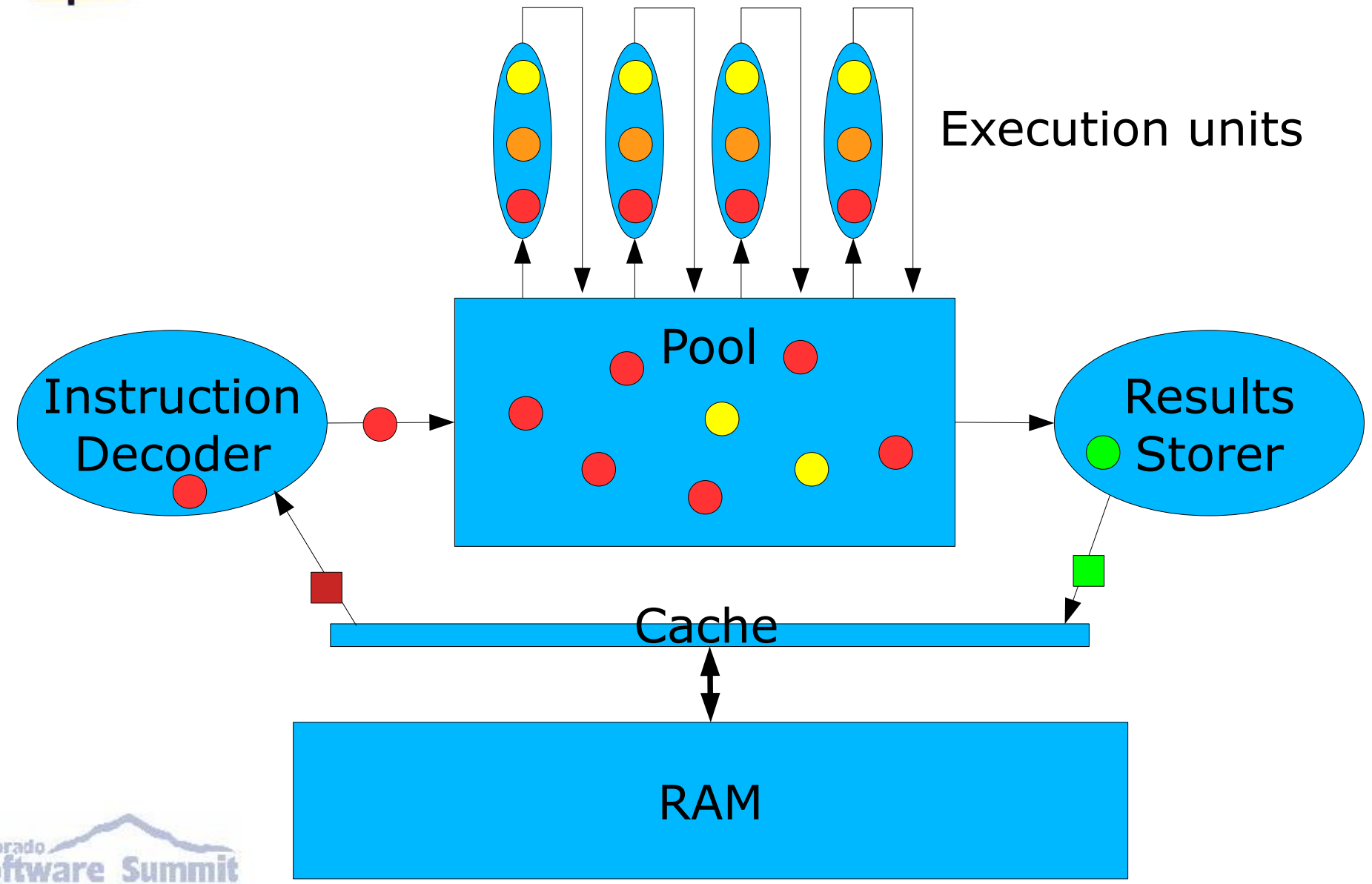
CPU Architecture

- Some instructions may take several clock cycles to complete, moving through several stages of computation.
  - Modern CPUs use "pipelining" to maximize silicon usage.
  - Instructions enter the pipeline and get progressively processed as they move to the end.
  - This means that one instruction can be processed per clock cycle.
  - However, dependencies can stall the pipe.

CPU Architecture

- Modern chips have masses of space for circuitry.
  - CPU manufacturers want to use it to speed up the CPU.
  - They invented a way to execute more than one instruction per clock cycle!
  - The CPU contains an "instruction pool". A decoder puts instructions in. The pool works out which are ready to execute, and passes them to multiple execution units. A unit finalises the instructions and stores the results.

CPU Architecture



CPU Architecture

- Instructions may be executed out of order.
  - However an instruction cannot be executed before the values it uses have been produced by previous instructions.
  - Instructions will stay in the pool until then.
  - Registers are renamed so that two separate uses of a register at different points in the code do not interfere with each other.
  - Depending on the code, usually manages to keep quite a few execution units busy.
  - Athlon had seven execution units.

Simultaneous Scans

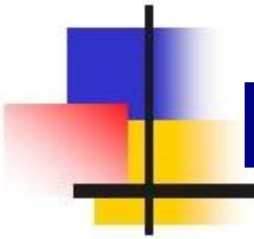
- PostgreSQL 8.3 introduced a new feature – simultaneous sequential scans.
  - If a query requires the sequential scan of a table, it checks to see if another query is already scanning that table.
  - If so, then it starts its scan from where the existing query is in the table, and shares its disc access. When it gets to the end of the table, it starts from the beginning and goes on to where it started.
  - Reduces disc bandwidth used.

Linux 2.6.25 scheduler

- When Linux kernel 2.6.25 came out, it incorporated a new task scheduler, designed to be fairer.
  - Unfortunately, it reduced the performance of some multi-threaded programs.
  - It switched between threads when they blocked on IO in a much fairer fashion.
  - Each thread next got CPU much later, so its working set had dropped out of CPU caches.
  - A little unfairness can help!

Linux Anticipatory IO

- Linux provides several disc IO schedulers, one of which is Anticipatory, designed for small systems.
  - Designed to reduce the impact of two threads accessing disc simultaneously.
    - Process 1 would request a block, and this would be dispatched to the disc. Process 1 sleeps.
    - Process 2 gets CPU time, and requests a block.
    - The disc gets the block for process 1, and the request from process 2 would be dispatched.
    - Process 1 gets CPU time, and asks for another block.



Linux Anticipatory IO

- The result was thrashing – alternately fetching a block from either end of the disc.
  - But both processes were doing sequential scans!
  - Readahead helps a little.
- The anticipatory IO scheduler would actually wait a little while before dispatching the request from process 2.
  - It hoped that a little bit of CPU time would let process 1 request another block very close to the last one.
- Big performance improvement sometimes!
- A little laziness can speed things up.

Linux Anticipatory IO

Selective Scan Buffers

- When PostgreSQL performs a sequential scan, it uses a very small buffer.
  - This keeps a sequential scan from pushing valuable stuff out of memory.
  - Also, a small buffer can fit in the CPU cache, resulting in the code running 25% faster.

Journalling

- Technique primary for providing crash-recovery on filesystems.
  - Can help with performance too.
  - When a write needs to be made, write the change to the end of a "journal" on disc.
  - The write is now safe. The changes can be written to the correct part of disc later.
  - Performance can be better because all the writes go to the same place on disc.
    - However, data is written more than once.
  - On crash recovery, replay the log.

Linux Network Stack Attacks

- Early in Linux's development, a way to make it slow down was found.
  - Another computer would send lots of TCP SYN packets over the network.
  - Linux would place these in a hash table and send a reply packet, then wait for a second packet with the right sequence.
  - However, all the SYN packets were tweaked to go into the same hash bucket.
  - Hash table went from **O(1)** to **O(n)**.
    - Linux would spent lots of time in kernel.

Linux Network Stack Attacks

- The attack would make a Linux machine unresponsive, and didn't require much network bandwidth.
  - A solution was found: When Linux boots up, it generates a random number, and keeps it secret.
  - This random number is used in the hashing function for the hash table.
  - Since the attacker doesn't know the random number, it cannot make all the packets go in the same hash bucket.

Linux Network Stack Attacks

- Another solution was found:
  - The purpose of the hash table was to store details of connections that were being made.
    - A sequence number from the remote host and a sequence number from the local host.
  - The local sequence was randomly generated.
  - Instead, generate the local sequence number from the remote sequence number using a one-way function with a secret key.
  - Then, the details need not be stored at all!

Acknowledgments

The FlyMine team: Richard Smith, Matthew Wakeling, Xavier Watkins, Julie Sullivan, Jakub Kaluviak, Hilde Janssens, Rachel Lyne, Dan Tomlinson, and Gos Micklem

The InterMine system (www.intermine.org) is a generic object-oriented database and data integration system, open source and licensed under the LGPL.

FlyMine (www.flymine.org) is a biology-specific application of the InterMine system, also licensed under the LGPL.

FlyMine is funded by the Wellcome Trust.

FlyMine Group, Cambridge Systems Biology Centre, University of Cambridge, Tennis Court Road, Cambridge, CB2 1QR, UK
Tel: +44 1223 760262 Email: info@flymine.org