Algorithms and Data Structures: Mid-term Review

What have we covered?

topic	Data Structures and Algorithms
intro & data abstraction	***APIs, ADTs, stack, queue, bag, union-find, O(N), entropy, proof by induction***
sorting	***merge sort, heap sort (priority queues), quicksort***
searching	symbol tables (maps), binary search trees, hash tables
graphs	undirected and directed graphs, minimum spanning trees, shortest path
advanced topics	strings, tries, substring search, regex, Btrees

Algorithms and data structures

- So, practically speaking, what are algorithms and data structures?
  - A data structure is an orderly collection of memory locations for the purpose of storing information for short or long-term—In the short-term, a data structure is typically used to support an algorithm.
  - An algorithm is a sequence of steps each of which manipulates a data structure to achieve some new state or result.

Why are A&DS important?

- Languages come and go; there are flavors of language which are best-suited to a particular type of application. Some languages are “better” than others: easier to write; easier to read; produce more efficient code; compile/interpret faster; higher/lower “level”; more or less verbose; etc. etc.

- But algorithms and data structures are **fundamental** to solving problems—for this reason, they haven’t changed very much in the last forty or fifty years.

- There are two precious commodities in any computer system: cycles (i.e. time) and memory (i.e. space). Good algorithms and data structures aim to minimize the use of these commodities.

Complexity and performance

- I used to have a colleague who was fond of saying: “There’s nothing like hardware to improve software.”

- Why is this a very bad thing to say?
  - Because hardware gains tend to be *linear* while software problems tend to be *polynomial*.

List, arrays, hash tables

	Size	Access	Construction
Array	Fixed	Random, by index	Random, by index
List	Variable	Sequential, starting from the head	Sequential, starting from the head
Hash Table	Growable	By key	By key and its hash

Comparison of storage methods

Technique	Access	Add to Head	Add to Tail	Copy
Array	O(1)	O(1)*	O(1)*	O(N)
Linked List	O(N)/2	O(1)	O(N)	O(1)
Doubly-linked List	O(N)/2	O(1)	O(1)	O(1)

* Except when full: in which case, a Copy is required

Primitives vs. Objects

- As well as the four primitives described above, there are also long (64-bit), short (16-bit), byte (8-bit) and (32-bit) float.
- All of these primitives have a “boxed” object which essentially wraps the primitive value: viz. Integer, Double, Long, Boolean, Char, etc. Boxing/unboxing is, normally, automatic (like widening) according to the context.
- Null pointers: If *x* is an object, you can write `if (x==null)`. If *x* is a primitive, this doesn’t make sense.
- Primitives and objects are all passed into methods by *value* in Java. That’s clear for a primitive—but what does it mean for an object? Objects are really just pointers.

The inner loop

- You nearly always have to worry only about the inner loop because, assuming *N* is large, the inner loop swamps the times of the other loops;
- So, for example, you have time to unpack the rows and columns of the two multiplicands from a less efficient form such as *List<List<Double>>* into *double[][]*.

Sort once; search many

- Back to our “dictionary principle” from earlier:
  - In order for the binary search algorithm to work, the array must be sorted;
  - This takes time, but it is only sorted once (at the beginning of the main program) while it can be searched almost an infinite number of times (the length of the input file).

Abstract Data Types

- Last week, I referred to ADTs and APIs without definitions. So…
  - *Data types*: A data type is a set of values and a set of operations on those values.
  - *Abstract data types*: An abstract data type is a data type whose internal representation is hidden from the client (encapsulation).
  - *Objects*: An object is an entity that can take on a data-type value. Objects are characterized by three essential properties: The state of an object is a value from its data type; the identity of an object distinguishes one object from another; the behavior of an object is the effect of data-type operations. In Java, a reference is a mechanism for accessing an object.
- *Applications programming interface (API)*: To specify the behavior of an abstract data type, we use an application programming interface (API), which is a list of constructors and instance methods (operations), with an informal description of the effect of each.
  - *Client*: A client is a program that uses a data type.
  - *Implementation*: An implementation is the code that implements the data type specified in an API.

Or, no internal representation at all

Equality and hashCode

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the *references* are equal. We also need a way to test logical or datatype equality. This is defined in the method *equals()*. When we define our own data types we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*:

- *Reflexive*: `x.equals(x)` is true.
- *Symmetric*: `x.equals(y)` is true if and only if `y.equals(x)` is true.
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are true, then so is `x.equals(z)`.

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns false

Must be consistent with *hashCode()*

Total order

Goal. Sort any type of data (for which sorting is well defined).

A total order is a binary relation $\leq$ that satisfies:

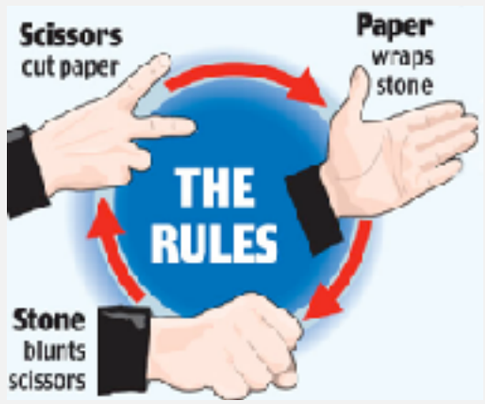
- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Ex.

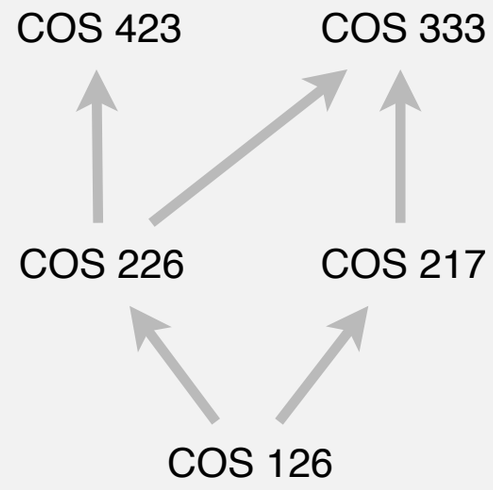
- Standard order for natural and real numbers.
- Chronological order for dates or times.
- Alphabetical order for strings.

No transitivity. Rock-paper-scissors.

No totality. PU course prerequisites.



violates transitivity



violates totality

Where does that *n lg n* thing actually come from?

- A list of *n* items can be in any one of *n!* permutations.
  - If it isn't sorted, you don't know which one of those it's in (it could be sorted by chance but you wouldn't know it). Therefore, the entropy of the unsorted list is, according to Shannon's definition, *—lg (1/n!)* or *lg n!*
- The standard method to estimate *n!* is Stirling's formula:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

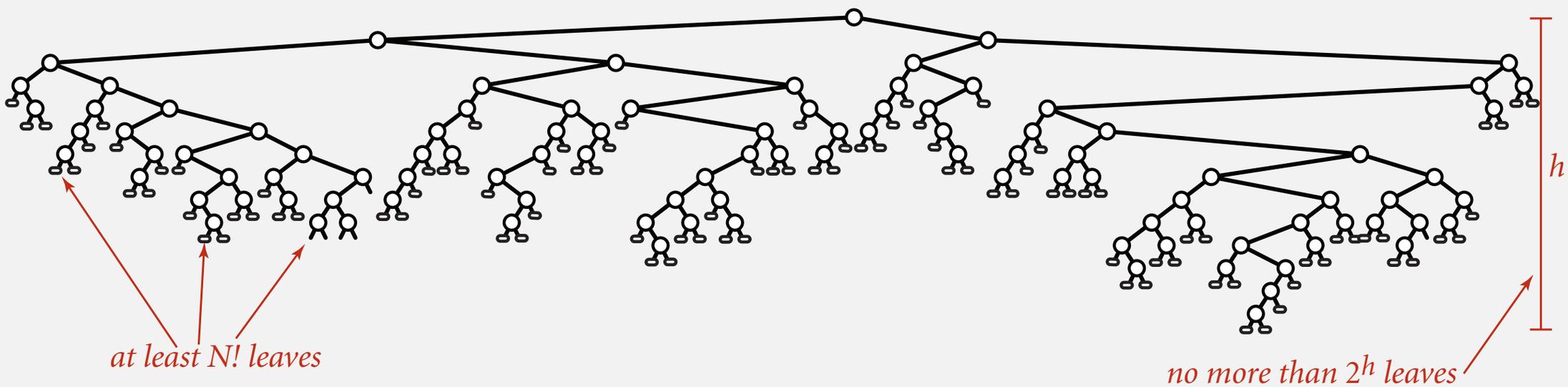
- As you can see, this is *exponential.* Not nice.
- Entropy, (in bits) is therefore: *~ n(lg n - lg e) + 1/2 lg (2πn)*
  - i.e. *~ n lg n*
    - We should therefore hope to find an algorithm that is O(*n lg n*) because every compare/swap should, ideally, remove one bit of entropy.
    - [note that we ignore the other terms because they aren't dominant for large n]

Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg(N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of N distinct values a_1 through a_N.
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.



Selection sort: Java implementation

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Comparable interface: review

Comparable interface: sort using a type's natural order.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }
   …
public int compareTo(Date that)
{
   if (this.year  < that.year ) return -1;
   if (this.year  > that.year ) return +1;
   if (this.month < that.month) return -1;
   if (this.month > that.month) return +1;
   if (this.day   < that.day  ) return -1;
   if (this.day   > that.day  ) return +1;
   return 0;
}
```

natural order

Comparison of simple sorts

Algorithm applied to random data	Comparisons	Swaps	Copies	Total Ops
Bubble*	$N^2/2$	$N^2/4$		$3N^2/4$
Selection	$N^2/2$	N		$N^2/2$
Insertion*	$N^2/4$		$N^2/4$	$N^2/2$

* Bubble and Insertion sorts are O(N) when the data is already sorted

Basics, entropy, scientific method

- Recall how to conduct a “doubling” experiment and how to predict performance from this.
- APIs, ADTs, equality, Java primitives/objects.

Bags, Stacks, Queues

- You need to recall the basics of these data structures.

Union-Find

- We looked at some simple strategies (quickFind and quickUnion) but these were lacking in performance: NM

- As with practically all simple algorithms, we found ways to make incremental improvements:

  - Weighted quick union to balance the trees:

    - *N + M lg N*

  - Weighting and path compression to keep the tree depth as low as possible: *N + M lg* N* where *lg** is that strange recursive definition of *lg*. Almost as good as N + M.

Simple Sorts, etc.

- You should be familiar with the basics of the sorts we've already looked at:
  - Insertion, selection sort;
  - Shell sort (will not be examined in mid-term)
  - Merge sort and Timsort
  - Quick sort and quick select
  - Heap sort and priority queues (next week)