 Fundamental Algorithms: Intro to the Class

Pragmatics
Questions
What it will cover
How it will cover it — Java
Efficiency
Pragmatics

Instructor

Brook Conner, conner@cs.nyu.edu, me@nellardo.com
Office hours 5-6 Wednesdays or by appt. Room 401.

Grader/TA

Aristotelis Tsirigos (tsirigos@cs.nyu.edu)

Email list

http://www.cs.nyu.edu/mailman/listinfo/g22_1170_001_su02

Texts

Policies
Texts

Required


"The Big White Book" (though it is now green)

Recommended

*Concrete Mathematics*, Graham, Knuth, Patashnik.

If your discrete mathematics is rusty....

*The Java Programming Language*, Gosling.

*Java in a Nutshell*, Flanagan.

If you’re new to Java....
Policies

Sign up for the email list

Weekly assignments
  Will be on website by start of class
  Due at start of class following week
  Prose answers on paper
  Executable code in email

Discussion of topics is cool; Discussion of answers is not

Asking questions in class is encouraged
Algorithms

“Algorithm” in dictionary looks like a recipe
But even recipe needs ingredients and tools
Really “Algorithms and Data Structures”
Can’t do one without the other
Maybe should be “Data systems”
But everyone calls them “algorithms” so we will too
Categories of Algorithms

Ordered Collections
  Transformations between different orders

Unordered Collections
  Sets, bags

Partially Ordered Collections
  Heaps, graphs

Embedded Collections
  Computational Geometry
Ordered Collections

Categorize by access methods

- First-in, first-out: Queue
- First-in, last-out: Stack
- Indexed random access: Array, List

Transform one ordered collection into another

- Priority queues
- Call stacks
- Permutations: Sorting, Fourier transforms, Wavelet transformations
Unordered Collections

Categorized by element multiplicity

Set: Every elt is unique

Bag: Not necessarily

Dictionary: A set with keys identifying the elements
    Hash table

Hey, where are trees?
Partially Ordered Collections

Heaps

- Everything on one layer is “above” everything below
- No ordering relative to same layer

Graphs

- Not USA Today or Excel
- Set of “nodes”
- Set of pairs of nodes — “edges”
  - Sometimes a bag of pairs
  - Sometimes ordered pairs (as in \((A, B) \neq (B, A)\))

Trees are graphs s.t. exists unique path everywhere except for “root”
Embedding

Where are the nodes located? Haven’t said yet

Come up with a point (or shape) for each node and a line (or curve) s.t. some useful property is true

   E.g., No edges cross (a planar graph)
   E.g., Minimum surface area of boundaries

Find interesting embedded graphs based on existing embedded graph

   E.g., Convex hull (the outermost nodes and edges)
   E.g., Closest pairs
   E.g., Voronoi diagram (divide space into points closer to one embedded node than any other)
Why Java?

You almost certainly know C, so syntax is palatable

You probably know one of C++ or Java

No memory management!

  If you don’t think this is a big win, you don’t know what you’re missing

Generally well-behaved enough for mathematic proofs of properties

Some useful language features for our purposes
Classes and Instances

Object: A mix of data and code

Class: A category of objects

Instance: A particular object with a particular class

```java
class ConsCell {
    public Object head;
    public ConsCell tail;
    public ConsCell(Object h, ConsCell t) {
        head = h; tail = t;
    }
    public Object car() { return head; }
    public ConsCell cdr() { return tail; }
    public Object cdar() { if (tail) {
        return tail.head;
    } else {
        throw new NullPointerException("No tail");
    }
}
```
Inheritance

Specialize a category

```java
class Animal { .... }
class Cat extends Animal { .... }
```

May only extend one class (single inheritance)

May implement multiple interfaces

```java
interface Carnivore { .... }
class Dog extends Animal implements Carnivore {
    ...
}
class Alligator extends Animal implements Carnivore {
    ....
}
```

This is where we come in....
Collections and Java

Interfaces to define collection categories

Classes to define implementations

Interfaces to define collection elements

Don’t much care about element implementation

   Just care about knowing exactly what we need to be able to do with them

   For you C++ programmers: No templates (and we don’t need them)
Example: Ordered List

```java
interface Comparable {
    public Comparable lessThan(Comparable);
}

interface OrderedList {
    public Comparable nthElement(int n);
    public void addElement(Comparable);
}

class ArrayOrderedList implements OrderedList {
    protected Comparable elts[];
    public Comparable nthElement(int n) { .... }
    public void addElement(Comparable c) { .... }
}
```
Efficiency

The von Neumann Architecture

Space-Time Tradeoffs

Little-o and Big-O
The von Neumann Architecture

Want to quantify performance

   Clock time, actual ram are both platform-specific

Abstraction!

State machine (i.e., CPU)

   Count transitions
   Extend to operations with no loss of generality

Memory (RAM)

   Access any value in one operation
   Cf. Turing machine: n steps to reach memory n locations away
Space or Time?

Efficient in memory or efficient in operations

Often, can trade one for the other

How much space and time?

Don’t want to measure a specific data set
Pathological cases exist
We want guarantees

More abstraction: Problem size is a variable, e.g., \( n \)

Collections: \( n \) is typically number of elements
Graphs: \( n \) is number of nodes, \( m \) is number of edges
Little o and Big O

Count operations and/or memory as a f(n)

Definition: \( o(3n^2 + 17n) \) means exactly that many operations/memory locations

Use a big O, and we have \( O(1) = O(c) \) for any \( c \)
Next Week

Assignment: See web site

Reading:

CLRS, Chapters 1–5. Intro, basic sorting, asymptotics, recurrences, randomization

Optional: GKP, Chapter 1, 9