# CS1003/1004: Intro to CS, Spring 2004

Lecture #9: Midterm review, data structures

Janak J Parekh
janak@cs.columbia.edu

## Administrivia

- HW#3 due now
- HW#4 out today
  - Less programming, more written
  - Some programming based on HW#3; I'll release solutions you can work off of if you want
- Midterms returned now

## Midterm statistics

| | CS1003 | CS1004 |
|---|---|---|
| Count | 26 | 46 |
| Mean | 38.15 | 37.43 |
| StDev | 8.44 | 8.61 |
| High | 49 | 50 |
| Low | 23 | 17 |

## How I grade?

- Grades added up *at end of semester* and then "scaled" appropriately
- Median grade in the class is borderline B/B+
- Remember, class participation helps
- Marked improvement also helps
- Come talk to me if you have any questions

## Midterm answers

- Part 1
  - CS1003: F, T, F, T, F
  - CS1004: F, F, T, T, F
  - I allowed partial credit, though
- Part 2, Q1
  - Algorithm finds *top two* numbers
  - Removing italics => val2 no longer is the second-highest number
  - O(n) algorithm

## Midterm answers cont'd.

- Part 2, Q2
  - 46 and 23
  - Dropping the last bit does integer division by two
- Part 2, Q3 – runs *9* times (i=1 through i=9)

```
int i = 1;
while(i < 10) {
  System.out.println(i); or printf("%d\n", i);
  i++;
}
```

## Midterm answers cont'd.

- Part 3: Note that prime #s start at 2!

```
int nextPrime = 2, numPrimes = 0;
while(numPrimes < n) {
  if(isPrime(nextPrime)) {
    print(nextPrime);
    numPrimes++;
  }
  nextPrime++;
}
```

## Why HW#3?

- I know it was a large programming assignment, but it was a necessary one
- In essence, summarized the "first half" of the semester
- You need these skills under your belt for the rest of the semester
- If you didn't quite finish, take a look at solutions, come to office hours, etc. and *make sure you understand*

## Bubble sort, reviewed

```
for(i=alength - 1; i > 0; i--) {
  for(j = 0; j < i; j++) {
    if(a[j] > a[j+1]) {
      int temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
    }
  }
}
```

- Why is this $O(n^2)$?

## Insertion sort

- Similar to bubble sort; *slightly* more efficient
- Principle: consider the left side the "sorted" side, and the right side the "unsorted" side
- Successively insert the "next unsorted" element into position into the "sorted" side
- Applets demoing this and Bubble sort: http://home.janak.net/cs3134/lafore-applets/Chap03/
- You can use either sort…

## Data structures

- We've been referring to this informally, but now let's be precise
- A computer's memory is a large open space, and we can organize information in it
- A *data structure* is an organized entity in this memory space
- The most primitive data structures: *primitive types*

## Primitive types

- int, char, double, etc.
- Occupy a well-known amount of memory
  - For 32-bit machines, an char takes *1 byte*, an int takes *4 bytes*, a double takes *8 bytes*
  - *Not always the case*, but enough for this class
- The variable refers to that block of memory in its entirety
  - Can't typically store decimal places inside an int; "won't fit"
- But what if we want something more complicated?

## Arrays

- I've arbitrarily defined these as a block of memory divided into cells
- To be more precise, an array is a *static* structure in memory
  - Memory is organized "contiguously" when you define an array
    - 10 integers => 10 * 4 => 40 bytes on a 32-bit machine
  - The variable referring to the array actually just points to the *beginning* of the appropriate memory location

## Arrays (2)

- The programming language then does some math when you use [ ] to access an index in that array…
  - An array of integers, length 10 is at memory location "4000".
  - How many bytes is this array in total?
  - What's the position of the 5th integer?
  - Rationale for 0-based makes a little more sense

## More generally…

- For primitive datatypes (int, char, etc.), the variable refers to that entity *in its entirety*
- But whenever we work with a more complex data structure than just a primitive datatype, our variable will "point" to the beginning of the structure
  - Known as a *pointer* (C) or a *reference* (Java)
- The programming language then decides what part of the memory starting at the variable you're working with

## Strings

- Strings are an interesting case
- In C, Strings are just arrays, and we treat them as blocks of memory of predefined size
- In Java, Strings are *dynamic*, and can vary in length
  - We'll get into more technical details later
- Here's why doing == with Strings doesn't work, though…

## Custom data types

- Wouldn't it be nice for HW#3 to have a single "entity" to refer to bank account, so we can have an array of *bank accounts* instead of two separate arrays?
- We can declare such a *structure* (C) or *object* (Java)
  - We'll set it up so that it contains a String and a double
  - We then access *components* of that "bank account"
- This week's lab will start with the basics on how to do exactly this

## How complicated?

- Data structures & types can be almost as complicated as you want
- You can nest complex data structures
  - For example, a bank account can contain an array of dependents
  - You can have an array of bank accounts in a Branch
  - You can have an array of Branches in a BankInstitution
  - And so on…
- How can we organize all this stuff!?
  - Take CS3134, and you'll learn all the details. Here's a few.

## "List" data abstraction

- The most common way to organize things is in a list
  - An array is one type of a list – it's *static* sizewise; "contiguous list"
- What are basic *conceptual* operations on a list?
- Can we organize lists in any different fashion?

## Next time

- Continue discussion on data structures