CSE 241 Class 1

Jeremy Buhler

August 24, 2015

Before class, write URL on board:

`http://classes.engineering.wustl.edu/cse241/`.

Also: Jeremy Buhler, Office: Jolley 506, 314-935-6180

1 Welcome and Introduction

Welcome to CSE 241: Algorithms and Data Structures!

- I'm your instructor, Dr. Jeremy Buhler.
  - I study computational biology
  - we deal with lots of classic algorithmic problems: searching for things, finding similar objects in a large set, etc.
  - "objects" are sequences of DNA and protein, and they occur in sets millions of bases in size. Specialized algos deal with them, though many rely on basic 241 stuff (you'll see an example!)
  - ... theorist some days (algorithm producer)
  - "informed consumer" too – basic searching, sorting, etc must run fast and not take much space, or biologists get grumpy!
  - I want to get you **excited**, or at very least **informed** about algos, so that you too can be a producer and/or informed consumer and make your colleagues happy!
- Who are you? *Ask for show of hands: CSE students? Engineering? Arts and Sciences? Sophomores? CS minors? Second majors?*

2 First, Some Administrivia (Course Policies)

Let's get this out of the way so we can proceed to the fun stuff!

1. First, everything I'm going to say and more is on the web site.
2. Office hours: Mon 10:30-12:00 Wed 10:30-12
3. TAs: lots of them; they will also hold office hours.

4. We will use **Piazza** for discussions and help requests. Please create an account if you have not done so and enroll in 241 to see the discussion boards.

5. TAs and I may not answer help requests sent via email, or we may ask you to repost via Piazza. With almost 200 people, it will be much easier to keep track if we keep all discussions centralized.

6. Evaluation: 5 homeworks worth 20%, 4 labs worth 20%, three equally weighted exams worth 60%. **No cumulative final!**

7. Turn-ins: no late homeworks accepted! (I'll try to hand out solutions immediately). Late labs up to three days, 10% off per day. If no late labs, can redo one and turn in at end of semester for regrade (cost of 10%).

8. We've implemented electronic turn-in for labs using personalized SVN repositories, similar to 131, and for homeworks using Blackboard. See the web site for instructions. Lab 0 and Homework 0 are out today, due 9/2, to make sure everyone can successfully use the infrastructure. (Yes, they are for credit.)

9. Collaboration: I try to balance fun with fairness. No written sharing, "Iron Chef rule". Everyone must read policy on class web and sign statement of compliance with each homework. Also, we plan to do electronic comparisons among turned-in labs to check for code copying.

10. **We will try to treat you with respect; please reciprocate.**

3 Goals, and a First Problem

What do we want you to learn?

- A way to **think about computation**. What is a "good" algorithm? Which of two algorithms is "better"? What does "fast"/"faster" mean?
- **Abstractions**: ways of thinking about how to organize data. You already know some – lists, queues, stacks. We'll introduce more abstractions – sparse tables, trees, graphs – that can be realized efficiently.
- A **"bag of tricks"**: basic algorithmic techniques – sorting, searching, data management – and analytical techniques – recursion trees, asymptotic notation, lower bounds. Both practical *and* basic to computer science culture (yes, it's more than just writing code).

Let's start on the first of these three points with an example problem.

- Air traffic control center
- n airplanes appear as points on a 2D screen
- At any time, you must quickly be able to identify the *closest pair of planes* (points) on the screen! (E.g., to sound a warning if they're too close).
- The airport is busy – many planes on screen at any time.

- *Closest pair problem*: other uses as well

[Draw the following picture here]

- **What is obvious solution to this problem?**
- For each point p, compute its distance to all other points q. (Write it:)

$$\sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$$

- Return the pair at minimum distance.

A little more formally...
Let $P[0 \ldots n-1]$ be an array of points.

```
CheckAll(P)
  minDist ← ∞
  j ← 0

  while j ≤ n − 2 do
    k ← j + 1
    while k ≤ n − 1 do
      d ← distance(P[j], P[k])
      if d < minDist
        minDist ← d
        p1 ← P[j]
        p2 ← P[k]
      k++
    j++

  return p1, p2, minDist
```

Does anyone have a better idea or improvement?
[Solicit, use as example of alternative algorithms.]

4 Is This a "Good" Algorithm?

- Is it **correct**? Despite what you may have heard, correct algorithms are always better than incorrect ones.

- (Yes, it's correct – checks every pair, stores pair of min distance.)
- "Correctness" means "correct for all possible inputs"!
- Is it **fast**? In particular, what if I gave you a different algorithm for closest pair – is CheckAll faster or slower?

How do we usefully measure speed of an algorithm?

- Code, debug, time execution? On what machine? With what inputs? Will the planes line up the same way as our test points? (If not, will we be sued for a crash?!?!?!?)
- We desire **machine independence**: say something useful about speed that doesn't depend on PC versus supercomputer versus smartphone
- Estimate statements executed in an "ideal" language, or on an "ideal machine"? Don't laugh – Turing machines, recursive function theory, Knuth's MIX architecture. We'll try this approach.
- Speed should be expressed as **function of input size**. Most interesting algorithms produce answers as a function of their inputs. The bigger the input, the more time to read it (and, in general, the more time to process it.) *Example*: more points n means CheckAll computes more distances.
- We need a **worst-case time estimate** for each input size: what if Dr. Evil designed the flight plan? What if we're unlucky?

  Make clear that choosing to consider the worst case is a fundamental assumption. Compare average case – can be useful if distribution of inputs known, but only if worst case is not catastrophic!

5 Let's Count Statements for CheckAll

Assume an input array P of size n.

```
minDist ← ∞
j ← 0
while j ≤ n − 2 do
    k ← j + 1
    while k ≤ n − 1 do
        d ← distance(P[j], P[k])
        if d < minDist
            minDist ← d
        k++
    j++
return minDist
```

Now let's add up all statements $S(n)$:

$$\begin{aligned} S(n) &= 3 + 3n - 2 + \sum_{j=0}^{n-2}(n-j) + 4\sum_{j=0}^{n-2}(n-j-1) \\ &= 3n + 1 + (n + n - 1 + \ldots + 2) + 4\,(n - 1 + n - 2 + \ldots + 1) \\ &= 3n + 1 + \left[\frac{n(n+1)}{2} - 1\right] + 4\left[\frac{(n-1)n}{2}\right] \\ &= \frac{5}{2}n^2 + \frac{3}{2}n \end{aligned}$$

Some hints for doing this yourself:

- For each loop in a nested set, write down the number of times its body statements execute as a function of its explicit bounds.
- (That would be high bound minus low bound plus one, if the interval is closed.)
- (For instance, the inner loop shown runs from $k = j + 1$ to $k = n - 1$, so its body executes $n - j - 1$ times. Wait until you've labeled every statment in this way before trying to sum over the different values of loop counters like j.)
- If a loop iterates m times, its test is executed $m+1$ times (including the last failure).

- Remember your sequence sums, e.g.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

6 Why It's OK to Approximate

Whew! That was a lot of work! The above approach gives a useful answer, but it is often **more precise than necessary**.

n	$5/2n^2$	$5/2n^2 + 3/2n$	% difference
10	250	265	5.7%
50	6250	6325	1.2%
100	25000	25150	0.6%
500	625000	625750	0.1%

- As you can see, **highest-order polynomial term of running time eventually dominates**.
- We say that this term determines the **worst-case asymptotic time complexity** of the algorithm.
- Hence, we say that running time is "roughly a constant times n^2," or, for short, $\Theta(n^2)$!!!! More on the meaning of Θ next week!
- "Eventually?" For large enough input size n, or, as computer scientists like to say, in **asymptopia** (a magical land where only asymptotic complexity matters).

What good is a simplified running time estimate?

- **Easier to compute.** Example: looking at CheckAll, it is "obvious" that the inner loop computes n choose 2 distances in the worst case for every n. This observation is enough to say that the algorithm's cost is *at least* a constant times n^2 (more on this next week).
- **Asymptotic comparisons are frequently meaningful in practice.** Suppose we had another closest-pair algo that runs in time $\Theta(n \log n)$? Even if the constant is moderately worse...

  [Show graphs of $T(n)$ versus n here. First graph shows quadratic algo winning, but point out small n. Now show other two graphs.]
- If you weren't sure how many planes might show up at the airport, which algorithm would you rather use?

Finally, a caveat about estimating running times by counting statements:

- In your "ideal language" of choice, be careful that statements take only constant time, regardless of input size.
- *Example*: In many languages, "list.length()" is a single statement, but its execution time depends on the size of the list.

- *Example*: Multiplying two matrices of size $n \times n$ is written as one line in Matlab but takes time $\Theta(n^3)$!

If you're careful, statement counts and their asymptotic estimates can be a valuable notion of what it means to have a "fast" algorithm.

[Show graphs of merge-sort versus insertion sort]