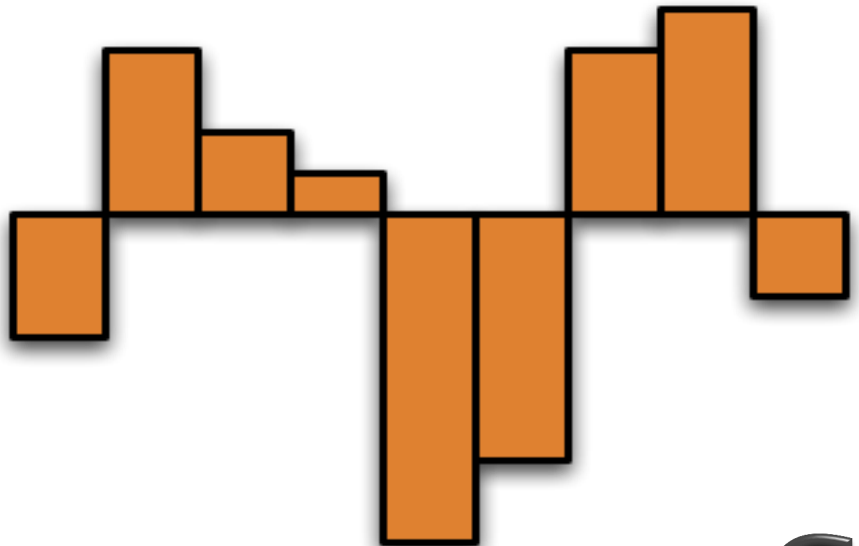
Q1–3,
4a



CSSE 230 Day 3

Maximum Contiguous Subsequence Sum
How to balance code simplicity and efficiency?

After today's class you will be able to:

- state and solve the MCSS problem on small arrays by observation
- find the exact runtimes of the naive MCSS algorithms

https://openclipart.org/image/2400px/svg_to_png/169467/bow_tie.png

Announcements

- Homework 1 due tonight
  - Lots of help available today if still working. Instructors, Lab TAs, CampusWire
- WarmUpAndStretching due after next class
  - Iterators? Read code comments, or Weiss Ch. 1–4.
- Reading for Day 4: Why Math?

Agenda and goals

- Finish up big-O, so you can
  - explain the meaning of big-O, big-Omega (Ω), and big-Theta (Θ)
  - apply the definition of big-O to asymptotically analyze functions, and running time of algorithms
- Analyze algorithms for a sample problem, Maximum Contiguous Subsequence Sum (MCSS), so you can
  - state and solve the MCSS problem on small arrays by observation
  - find the exact runtimes of the naive MCSS algorithms

Asymptotics: The “Big” Three

Big-O
Big-Omega
Big-Theta

Big-O, Big-Omega, Big-Theta

O( ) Ω( ) Θ( )

- f(n) is O(g(n)) if there exist c, n_0 such that:
  $$f(n) \le cg(n) \text{ for all } n \ge n_0$$
  - So big-Oh (O) gives an upper bound

- f(n) is Ω(g(n)) if there exist c, n_0 such that:
  $$f(n) \ge cg(n) \text{ for all } n \ge n_0$$
  - So big-omega (Ω) gives a lower bound

- f(n) is Θ(g(n)) if it is both O(g(n)) and Ω(g(n))
  Or equivalently:
- f(n) is Θ(g(n)) if there exist c_1, c_2, n_0 such that:
  $$c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$
  - So big-theta (Θ) gives a tight bound

Big-Oh Style

- Give tightest bound you can
  - Saying $3n + 2$ is $O(n^3)$ is true*, but not as precise as saying it's $O(n)$
  - *When we ask for true/false, use the definitions.
  - And when analyzing code, we'll just ask for Θ to be clear.

- Simplify:
  - You could also say: $3n + 2$ is $O(5n - 3\log(n) + 17)$
  - And it would be technically correct…
  - It would also be poor taste … and your grade will reflect that.

Uses of O, Ω, Θ

- By definition, applied to *functions.*

  "$f(n) = n^2/2 + n/2 - 1$ is $\Theta(n^2)$"

- Can also be applied to an *algorithm*, referencing its **running time**: e.g., when f(n) describes the number of executions of the most-executed line of code.

  "selection sort is $\Theta(n^2)$"

- Finally, can be applied to a *problem*, referencing its **complexity**: the running time of the best algorithm that solves it.

  "The sorting problem is $O(n^2)$"

Q6

Efficiency in context

- There are times when one might choose a higher-order algorithm over a lower-order one.

- Brainstorm some ideas to share with the class

C.A.R. Hoare, inventor of quicksort, wrote:

Premature optimization is the root of all evil.

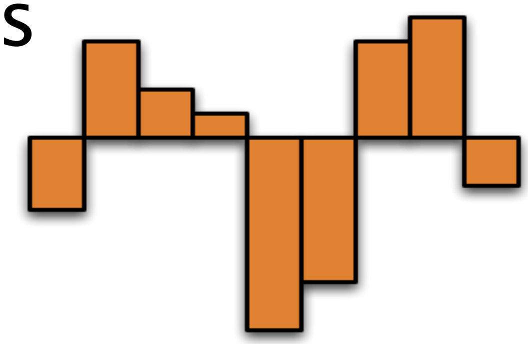
Maximum Contiguous Subsequence Sum

A deceptively deep problem
with a surprising solution.

{−3, 4, 2, 1, −8, −6, 4, 5, −2}

A Nice Algorithm Analysis Example

- **Problem**: Given a sequence of numbers, find the maximum sum of a contiguous subsequence.



- Why study?
- Positives and negatives make it interesting. Consider:
  - What if all the numbers were positive?
  - What if they all were negative?
  - What if we left out “contiguous”?
- Analysis of obvious solution is neat
- We can make it more efficient later.

Q7-10

Formal Definition of MCSS

- Problem definition: given a nonempty sequence of *n* (possibly negative) integers $A_0, A_1, A_2, \ldots, A_{n-1}$, find the maximum contiguous subsequence

$$S_{i,j} = \sum_{k=i}^{j} A_k$$

and the corresponding values of i and j.

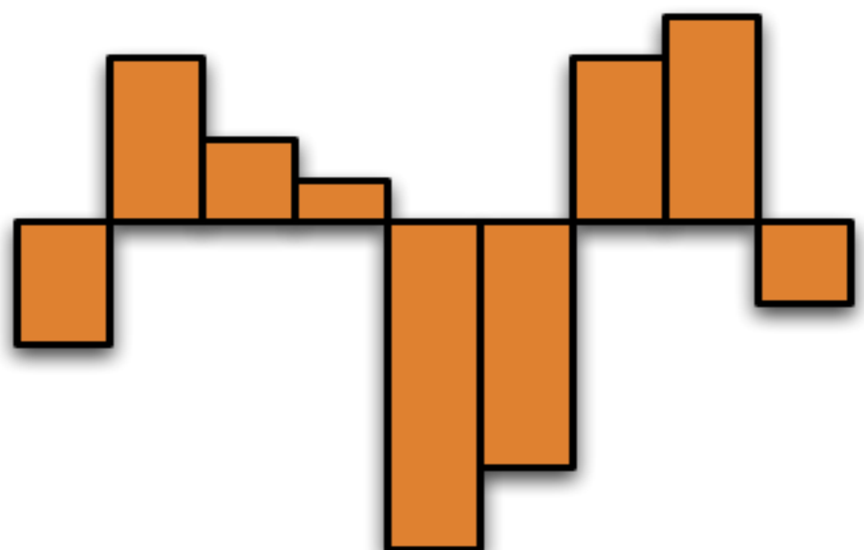
- Quiz questions:
  - In {−2, **11, −4, 13**, −5, 2}, $S_{1,3} = ?$
  - In {1, −3, 4, −2, −1, 6}, what is MCSS?
  - If every element is negative, what's the MCSS?

Write a simple correct algorithm now

Q11

- Must be easy to explain
- Correctness is KING. Efficiency doesn't matter yet.
- 3 minutes

- Examples to consider:
  - {-3, 4, 2, 1, -8, -6, 4, 5, -2}
  - {5, 6, -3, 2, 8, 4, -12, 7, 2}



First Algorithm

Find the sums of *all* subsequences

```
public final class MaxSubTest {
    private static int seqStart = 0;
    private static int seqEnd = 0;

    /* First maximum contiguous subsequence sum algorithm.
     * seqStart and seqEnd represent the actual best sequence.
     */
    public static int maxSubSum1( int [ ] a ) {
        int maxSum = 0;
            //In the analysis we use "n" as a shorthand for "a.length"
        for( int i = 0; i < a.length; i++ )
            for( int j = i; j < a.length; j++ ) {
                int thisSum = 0;

                for( int k = i; k <= j; k++ )
                    thisSum += a[ k ];

                if( thisSum > maxSum ) {
                    maxSum   = thisSum;
                    seqStart = i;
                    seqEnd   = j;
                }
            }
        return maxSum;
    }
```

i: beginning of subsequence

j: end of subsequence

k: steps through each element of subsequence

Where will this algorithm spend the most time?

How many times (exactly, as a function of N = a.length) will that statement execute?

Q12

Analysis of this Algorithm

- What statement is executed the most often?
- How many times?

```
for(int i = 0; i < a.length; i++) {
      for(int j = i; j < a.length; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                  thisSum += a[k];
            }
            // update max if thisSum is better
      }
}
```

Where do we stand?

- We showed MCSS is $O(n^3)$.
  - Showing that a **problem** is O(g(n)) is relatively easy – just analyze a known algorithm.
- Is MCSS $\Omega(n^3)$?
  - Showing that a **problem** is Ω (g(n)) is much tougher. How do you prove that it is impossible to solve a problem more quickly than you already can?
  - Or maybe we can find a faster algorithm?

f(n) is O(g(n)) if $f(n) \le cg(n)$ for all $n \ge n_0$
- So O gives an upper bound

f(n) is Ω(g(n)) if $f(n) \ge cg(n)$ for all $n \ge n_0$
- So Ω gives a lower bound

f(n) is θ(g(n)) if $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$
- So θ gives a tight bound
- f(n) is θ(g(n)) if it is both O(g(n)) **and** Ω(g(n))

What is the main source of the simple algorithm's inefficiency?

```
for(int i = 0; i < a.length; i++) {
      for(int j = i; j < a.length; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                  thisSum += a[k];
            }
            // update max if thisSum is better
      }
}
```

- The performance is bad!

Eliminate the most obvious inefficiency…

```
for(int i = 0; i < a.length; i++) {
      int thisSum = 0;
      for(int j = i; j < a.length; j++) {
            thisSum += a[j];
            // update max if thisSum is better
      }
}
```

- Remember the previous sum so we don't have to recompute it!

This is Θ(?)

MCSS is $O(n^2)$

- Is MCSS $\Omega(n^2)$?
  - Showing that a problem is Ω (g(n)) is much tougher. How do you prove that it is impossible to solve a problem more quickly than you already can?
  - Can we find a yet faster algorithm?

f(n) is O(g(n)) if $f(n) \leq cg(n)$ for all $n \geq n_0$
- So O gives an upper bound

f(n) is Ω(g(n)) if $f(n) \geq cg(n)$ for all $n \geq n_0$
- So Ω gives a lower bound

f(n) is θ(g(n)) if $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- So θ gives a tight bound
- f(n) is θ(g(n)) if it is both O(g(n)) **and** Ω(g(n))

Can we do even better?

Tune in next time for the exciting conclusion!